

THOMAS H. CORMEN
CHARLES E. LEISERSON
RONALD L. RIVEST



Twenty Second Printing

ALGORITHMS

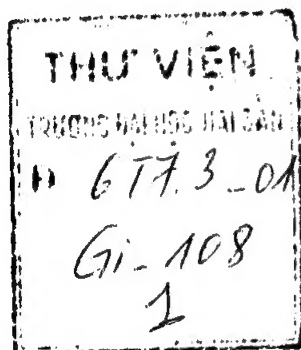
GIÁO TRÌNH
THUẬT TOÁN

Dành cho Sinh viên - Giáo viên - Chuyên gia

... dụ, Hình, hơn 900 Bài tập ... toán điển cứu có chọn lọc

NHÀ XUẤT BẢN THỐN...

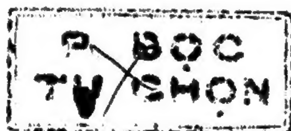
Chủ Biên : NGỌC ANH THƯ
Nhóm Biên Dịch : NGUYỄN TIẾN - NGUYỄN VĂN HOÀI
NGUYỄN HỮU BÌNH - ĐẶNG XUÂN HƯỜNG
NGÔ QUỐC VIỆT - TRƯƠNG NGỌC VÂN

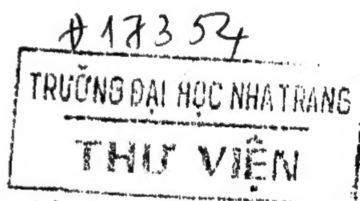


Giáo Trình
THUẬT TOÁN
LÝ THUYẾT VÀ BÀI TẬP

Sơ Cấp - Trung Cấp - Cao Cấp

- Giáo trình dành cho Sinh viên
- Tài liệu tham khảo giảng dạy của Giáo viên
- Cẩm nang thuật toán dành cho các chuyên gia
- Nhiều ví dụ, hình, 900 bài tập và hơn 120 bài toán để bạn trắc nghiệm mức tiếp thu căn bản về nội dung - tư duy tự kiểm tra





NHÀ XUẤT BẢN THỐNG KÊ

Nhóm **Ngọc Anh Thư Press**®

Trân Trọng Giới Thiệu Tới Bạn Đọc

Các Sách Sắp Xuất Bản

1. ***XML Nhập Môn - Thực Hành & Ứng Dụng***
2. ***Cấu Trúc Dữ Liệu & Giải Thuật Java***
3. ***Giáo Trình Lập Trình Hướng Đối Tượng Java***
4. ***Phần Cứng Máy Tính - Kỹ Thuật và Giải Pháp (Tập 1 và 2)***
5. ***Lập Trình C#***

33-333 / TK-2001 137 - 99 - 2001

Lời Nói Đầu

Có những cuốn sách viết về thuật toán tuy nghiêm túc song không đầy đủ và có những cuốn sách tuy nội dung phong phú song lại không nghiêm túc. *Giáo trình Thuật toán, Sơ cấp, Trung cấp, và Cao cấp* vừa mang tính toàn diện vừa đầy đủ. Có thể dùng nó làm sách giáo khoa, cẩm nang, sổ tay, và tài liệu tham khảo chung.

Giáo trình Thuật toán đề cập cả nội dung cổ điển lẫn các phát triển hiện đại như phân tích có khấu trừ và các thuật toán song song. Phần giải thích toán học, tuy nghiêm túc, song vẫn được nêu chi tiết cẩn thận để mọi tầng lớp độc giả có thể nắm vững. Các chương được tổ chức sao cho chúng bắt đầu với nội dung sơ cấp rồi tiến dần lên các chủ đề cao cấp hơn.

Mỗi chương tương đối độc lập và có thể dùng như một đơn vị học trình. Các thuật toán được trình bày theo dạng mã giả mà mọi người đã quen với Fortran, C, hoặc Pascal, đều có thể hiểu được. Vô số ví dụ có hình minh họa, hơn 900 bài tập, và 120 bài toán điển cứu thích đáng nhấn mạnh các khía cạnh toán học lẫn thiết kế kỹ thuật của chủ đề.

Giáo trình Thuật toán thật hữu ích dành cho các bạn Sinh viên từ khóa học năm cuối về các cấu trúc dữ liệu cho đến khóa học sau đại học về các thuật toán. Sách thực sự là một cuốn giáo trình trước mắt và một tài liệu toán học hoặc một cuốn cẩm nang thiết kế kỹ thuật luôn có mặt trên bàn giấy để bạn tham khảo cho nghề nghiệp của mình sau này.

Với các Giáo viên đang giảng dạy có thể thoải mái tổ chức khóa học dựa trên những chương cần thiết và được thiết kế tương đối độc lập với nhau, do đó giáo viên có thể lựa chọn chất liệu thích hợp nhất hỗ trợ cho khóa học mà mình sẽ dạy.

Với các Chuyên gia, sách với nhiều chủ đề đa dạng đã trở thành một cẩm nang tuyệt vời về thuật toán. Bởi hầu hết các thuật toán mô tả trong sách này đều có tính tiện ích cao, mô tả các phương án thay thế thực tiễn đối với một số thuật toán thiên về lý thuyết, cùng nhiều vấn đề hay, mới và thú vị dành cho bạn.

Chúng tôi xin trân trọng giới thiệu và rất mong sự đón nhận và động viên của các bạn đọc để tủ sách Ngọc Anh Thư Press của chúng tôi ngày càng tốt hơn.

Xuân Nhâm Ngọ - 2002
Ngọc Anh Thư Press(R)

Mục Lục

Phần III Các cấu trúc dữ liệu

Phần IV Các kỹ thuật phân tích và thiết kế cao cấp

VI *Thuật toán đồ thị*

Phần VII Các chủ đề chọn lọc

Lời tựa

Giáo trình Thuật toán—Lý thuyết và bài tập Sơ cấp - Trung cấp - Cao cấp là tập sách giới thiệu toàn diện công trình nghiên cứu hiện đại về các thuật toán máy tính. Không những trình bày nhiều thuật toán và đề cập khá chuyên sâu, nó còn mô tả cách thiết kế và phân tích chúng sao cho mọi tầng lớp độc giả đều có thể tiếp thu. Chúng tôi gắng giữ phần giải thích ở mức căn bản song vẫn không để mất tính chuyên sâu hoặc độ chính xác toán học.

Mỗi chương trình bày một thuật toán, một kỹ thuật thiết kế, một lĩnh vực ứng dụng, hoặc một chủ đề liên quan. Các thuật toán được mô tả bằng tiếng Anh và một "mã giả" [pseudocode] được thiết kế để những ai đã từng biết chút ít về lập trình đều có thể đọc được. Cuốn sách chứa đựng trên 260 hình, minh họa cách làm việc của các thuật toán. Do *tính hiệu quả* là một trong những quy chuẩn thiết kế của sách, nên chúng tôi đã tiến hành phân tích cẩn thận các góc độ thời gian thực hiện của tất cả các thuật toán.

Nội dung của sách chủ yếu được dùng cho các khóa học năm cuối cao đẳng/đại học và sau đại học về thuật toán hoặc cấu trúc dữ liệu. Do nó đề cập các vấn đề thiết kế kỹ thuật [engineering] trong thiết kế thuật toán, cũng như các lĩnh vực toán học, nên cũng thích hợp với các chuyên viên kỹ thuật muốn tự học.

Với thầy giáo

Cuốn sách này được thiết kế vừa linh hoạt vừa toàn diện. Nó sẽ hữu ích đối với bạn trong nhiều khóa học khác nhau, từ khóa học năm cuối về các cấu trúc dữ liệu cho đến khóa học sau đại học về các thuật toán. Do nội dung chuyển tải nhiều hơn đáng kể so với một đơn vị học trình thông thường, nên bạn có thể xem cuốn sách như một "tủ chén bát" hoặc "quầy thức ăn dọn sẵn" mà bạn có thể lựa chọn chất liệu thích hợp nhất hỗ trợ cho khóa học sẽ dạy.

Bạn có thể thoải mái tổ chức khóa học dựa trên chỉ những chương cần thiết. Các chương ở đây được thiết kế tương đối độc lập nhau, do đó bạn không cần bận tâm về khả năng lệ thuộc bất ngờ và không cần thiết giữa các chương. Mỗi chương đều đề cập nội dung theo trình tự dễ

trước khó sau, có các điểm ngừng tự nhiên đánh dấu ranh giới mỗi đoạn. Trong khóa học cuối cấp, bạn có thể chỉ dùng các đoạn đầu của một chương; trong các khóa cao học, bạn có thể đề cập nguyên cả chương.

Tập sách có gộp trên 900 bài tập và 120 bài toán. Mỗi đoạn kết thúc bằng các bài tập, và mỗi chương kết thúc bằng các bài toán. Nói chung, các bài tập là những câu hỏi ngắn trắc nghiệm mức tiếp thu căn bản về nội dung. Một số là bài tập tư duy tự kiểm tra đơn giản, trong khi một số khác lại thích hợp để ra bài tập ở nhà. Các bài toán là những điển cứu [case studies] công phu hơn, thường giới thiệu nội dung mới; nói chung, chúng gộp một vài câu hỏi dẫn dắt sinh viên qua các bước cần thiết để đi đến một giải pháp.

Chúng tôi đánh dấu sao (*) các đoạn và các bài tập thích hợp cho các sinh viên cao học hơn so với các sinh viên cuối cấp. Một đoạn có đánh dấu sao không nhất thiết khó hơn một đoạn bình thường, song nó có thể yêu cầu một trình độ toán học cao cấp hơn. Cũng vậy, các bài tập có đánh dấu sao có thể yêu cầu một vốn kiến thức cao cấp ngoài khả năng sáng tạo trung bình.

Với sinh viên

Hy vọng cuốn giáo trình này sẽ cung cấp cho bạn một số khái niệm hấp dẫn về lĩnh vực thuật toán. Chúng tôi đã cố gắng trình bày mọi thuật toán theo góc cạnh dễ tiếp thu và thú vị. Để giúp bạn khi gặp các thuật toán lạ hoặc khó, chúng được mô tả theo từng bước một. Tập sách cũng giải thích các khái niệm toán học cần thiết để giúp bạn nắm vững tiến trình phân tích các thuật toán. Nếu đã hơi quen với một chủ đề, bạn sẽ thấy các chương được tổ chức để bạn có thể lướt nhanh các đoạn giới thiệu và nhanh chóng bắt kịp các nội dung cao cấp hơn.

Đây là một cuốn sách lớn, và lớp học của bạn ắt chỉ đề cập một phần nội dung của nó. Tuy nhiên, chúng tôi đã gắng biến nó thành một cuốn sách hữu ích cho bạn với tư cách là một cuốn giáo trình trước mắt và một tài liệu toán học hoặc một cuốn cẩm nang thiết kế kỹ thuật luôn có mặt trên bàn giấy để bạn tham khảo cho nghề nghiệp của mình sau này.

Điều kiện tiên quyết để đọc cuốn sách này

- Để đạt kết quả cao, bạn phải có một số kinh nghiệm nhất định về lập trình. Đặc biệt, bạn phải hiểu rõ các thủ tục đệ quy và các cấu trúc dữ liệu đơn giản như các mảng và các danh sách nối kết [linked lists].
- Bạn phải có khả năng chứng minh bằng phép quy nạp toán học.

Một vài phần trong sách dựa vào một số kiến thức căn bản về tính toán. Ngoài ra, Phần I của cuốn sách này sẽ giải thích tất cả kỹ thuật

toán học mà bạn cần biết.

Với các chuyên gia

Với nhiều chủ đề đa dạng, cuốn sách này đã trở thành một cẩm nang tuyệt vời về thuật toán. Do mỗi chương tương đối độc lập với nhau, nên bạn có thể tập trung vào các chủ đề mà bạn quan tâm nhất.

Hầu hết các thuật toán mô tả trong sách này đều có tính tiện ích thực tiễn cao. Do đó, ta sẽ xét đến khả năng thực thi và các vấn đề thiết kế kỹ thuật khác. Nói chung, tập sách sẽ mô tả các phương án thay thế thực tiễn đối với một số thuật toán thiên về lý thuyết.

Nếu muốn thực thi một thuật toán nào đó, bạn có thể dễ dàng phiên dịch mã giả của tập sách này thành ngôn ngữ lập trình mà bạn ưa thích. Mã giả được thiết kế để trình bày từng thuật toán thật rõ ràng và súc tích. Kết quả là, tập sách không đề cập đến tính năng điều quản lỗi và các vấn đề thiết kế phần mềm khác, đòi hỏi phải có các giả thiết cụ thể về môi trường lập trình của bạn. Chúng tôi gắng trình bày đơn giản và trực tiếp từng thuật toán mà không nêu các đặc trưng của một ngôn ngữ lập trình cụ thể nào có thể làm lu mờ điều cốt lõi của nó.

Cáo lỗi

Một cuốn sách với số trang dày như thế này chắc chắn không thể tránh được các sai sót đáng tiếc. Chúng tôi xin thành thật mong quý bạn thông cảm và trông đợi sự góp ý xây dựng của quý bạn. Ngoài ra, chúng tôi cũng hân hoan đón nhận các gợi ý về các bài tập và các bài toán mới, song cũng làm ơn gộp phần đáp án. Bạn có thể gửi về cho chúng tôi thông qua Nhà Xuất bản Thống Kê và xin trân trọng cảm ơn.

Tập thể tác giả
NGỌC ANH THƯ
PRESS

Nhập Môn Thuật Toán

1 Giới Thiệu

Chương này giúp bạn làm quen với cơ trường hợp̉u sẽ được dùng suốt cuốn sách này để suy nghĩ về tiến trình thiết kế và phân tích các thuật toán. Tuy chương mang tính độc lập, song nó cũng gộp vài tham chiếu đến các nội dung sẽ được giới thiệu trong Phần I.

Để bắt đầu, ta tìm hiểu các vấn đề tính toán nói chung và các thuật toán cần thiết để giải quyết chúng, lấy bài toán sắp xếp làm ví dụ thực tiễn. Chương này cũng nêu một "mã giả" quen thuộc với những bạn đọc đã từng lập trình trên máy tính để nêu cách đặc tả các thuật toán. Sắp xếp chèn [insertion sort], một thuật toán sắp xếp đơn giản, được dùng làm ví dụ mở đầu. Ta sẽ phân tích thời gian thực hiện [running time] của tiến trình sắp xếp chèn, giới thiệu một hệ ký hiệu tập trung vào cách thức mà thời gian đó tăng lên theo số lượng các mục được sắp xếp. Ta cũng sẽ tìm hiểu cách tiếp cận "chia để trị" đối với tiến trình thiết kế các thuật toán và dùng nó để phát triển một thuật toán có tên sắp xếp trộn [merge sort]. Phần cuối chương sẽ so sánh hai thuật toán sắp xếp.

1.1 Thuật toán

Nôm na, ***thuật toán*** [algorithm] là một thủ tục tính toán được định nghĩa kỹ, sử dụng một giá trị, hoặc tập hợp các giá trị nào đó, làm ***đầu vào*** và cho ra một giá trị, hoặc tập hợp các giá trị nào đó, làm ***kết xuất***. Do đó, một thuật toán là một trình tự các bước tính toán biến đổi đầu vào thành kết xuất.

Cũng có thể xem một thuật toán như một công cụ để giải quyết một ***bài toán*** thật cụ thể. Phát biểu của bài toán sẽ chỉ định tổng quát mối quan hệ nhập/xuất cần thiết. Thuật toán mô tả một thủ tục tính toán cụ thể để đạt được mối quan hệ nhập/xuất đó.

Để bắt đầu quá trình nghiên cứu các thuật toán, ta sử dụng bài toán sắp xếp một dãy các con số theo thứ tự không giảm [nondecreasing]. Bài toán này thường nảy sinh trong thực tế và cũng là mảnh đất màu mỡ

để giới thiệu nhiều kỹ thuật thiết kế và các công cụ phân tích chuẩn. Dưới đây là cách định nghĩa hình thức ***bài toán sắp xếp:***

Đầu vào: Một dãy n số (a_1, a_2,..., a_n).

Kết xuất: Một phép hoán vị (sắp xếp lại) (a_1', a_2',..., a_n') của dãy đầu vào

sao cho $a_1' \leq a_2' \leq \ldots \leq a_n'$.

Cho một dãy đầu vào như (31, 41, 59, 26, 41, 58), một thuật toán sắp xếp sẽ trả một dãy kết xuất là (26, 31, 41, 41, 58, 59). Một dãy đầu vào như vậy được gọi là một ***minh dụ*** [instance] của bài toán sắp xếp. Nói chung, một ***minh dụ của một bài toán*** bao gồm tất cả mọi đầu vào (thỏa mọi hạn chế đã đề ra trong phát biểu của bài toán) cần thiết để tính toán một nghiệm cho bài toán.

Sắp xếp là một phép toán căn bản trong khoa học máy tính (nhiều chương trình dùng nó như một bước trung gian), và kết quả là nhiều thuật toán sắp xếp tốt đã được phát triển. Thuật toán nào là tốt nhất đối với một ứng dụng đã cho, điều này tùy thuộc vào số lượng các mục sẽ được sắp xếp, chừng mực mà các mục đó đã được sắp xếp sẵn, và loại thiết bị lưu trữ được dùng: bộ nhớ chính, các đĩa, hoặc băng từ.

Một thuật toán được xem là ***đúng đắn*** nếu, với mọi minh dụ đầu vào, nó ngừng với kết xuất đúng. Ta nói rằng một thuật toán đúng ***giải quyết*** được bài toán đã cho. Một thuật toán sai có thể không ngưng gì cả trên vài minh dụ đầu vào, hoặc có thể ngưng với đáp án ngoài ý muốn. Trái với dự đoán, các thuật toán sai đôi lúc cũng hữu ích, nếu như có thể kiểm soát mức độ lỗi của chúng. Ta sẽ thấy một ví dụ về điều này trong Chương 33 khi nghiên cứu các thuật toán để tìm các số nguyên tố lớn. Tuy nhiên, bình thường ta chỉ quan tâm đến các thuật toán đúng.

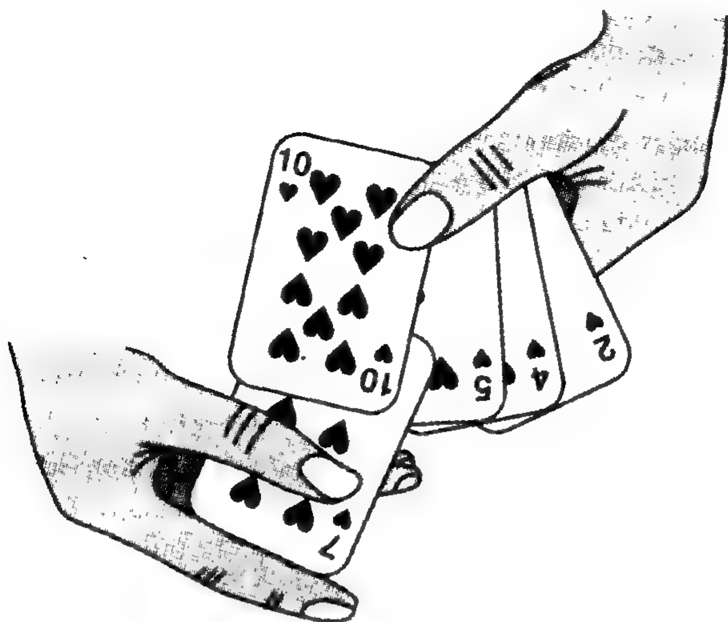
Một thuật toán có thể được đặc tả bằng tiếng Anh, như một chương trình máy tính, hoặc thậm chí như một thiết kế phần cứng. Yêu cầu duy nhất đó là phần đặc tả phải mô tả chính xác thủ tục tính toán sẽ phải theo.

Tập sách này sẽ mô tả các thuật toán dưới dạng các chương trình được viết bằng ***mã giả*** rất giống với C, Pascal, hay Algol. Nếu đã quen với một trong số các ngôn ngữ này, bạn có thể dễ dàng đọc các thuật toán ở đây. Sự khác biệt giữa mã giả và mã "thật" đó là: trong mã giả, ta sử dụng mọi phương pháp biểu đạt rõ ràng và súc tích nhất để đặc tả một thuật toán nhất định. Thỉnh thoảng, phương pháp rõ rệt nhất là tiếng Anh, do đó không có gì ngạc nhiên nếu bạn gặp một câu hay cụm từ tiếng Anh (*mà trong trường hợp thuận tiện chúng tôi sẽ sử dụng tiếng Việt để quý bạn tiện theo dõi*__ND) xen lẫn trong một

đoạn mã "thật". Một khác biệt nữa giữa mã giả và mã thật đó là: mã giả thường không dính dáng đến các vấn đề về thiết kế kỹ thuật phần mềm. Các vấn đề như trừu tượng hóa dữ liệu, tính môđun, và điều quản lỗi thường được bỏ qua để chuyển tải chính xác hơn nội dung cốt lõi của thuật toán.

Sắp xếp chèn

Trước tiên, ta tìm hiểu phương pháp ***sắp xếp chèn*** [insertion sort], đây là một thuật toán hiệu quả để sắp xếp các thành phần có số lượng nhỏ. Kỹ thuật sắp xếp chèn làm việc giống như cách thức mà nhiều người xếp một tay bài tây hay bài rumi. Ta bắt đầu bằng một tay trái trắng với các lá bài tung sấp trên bàn. Sau đó, lần lượt dỡ từng lá bài một ra khỏi bàn rồi chèn nó vào đúng vị trí trong tay trái. Để tìm đúng vị trí cho một lá bài, ta so sánh nó với mỗi lá bài sẵn có trong tay, từ phải qua trái, như minh họa trong Hình 1.1.

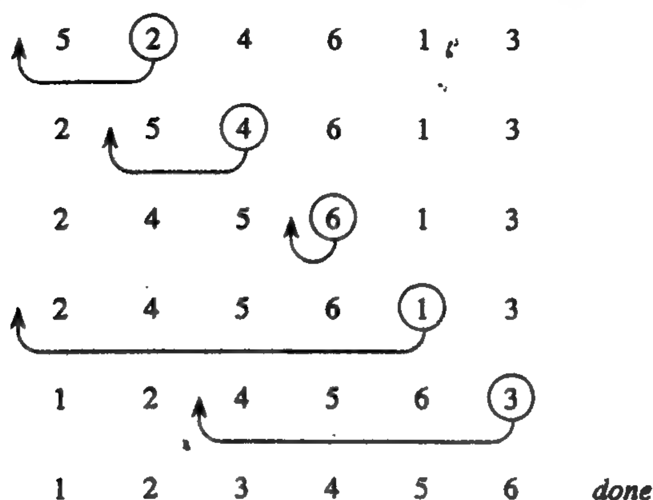


Hình 1.1 Xếp một tay bài theo phương pháp sắp xếp chèn.

Mã giả sắp xếp chèn của chúng ta được trình bày dưới dạng một thủ tục tên INSERTION-SORT, nó nhận một mảng $A[1..n]$ dưới dạng một tham số, chứa một dãy có chiều dài n sẽ được sắp xếp. (Trong mã, số n thành phần trong A được biểu thị bằng $length[A]$.) Các con số nhập được ***sắp xếp tại chỗ***: các con số được dàn xếp lại trong mảng A, luôn có tối đa một số lượng bất biến của chúng được lưu trữ ngoài mảng. Mảng đầu vào A chứa dãy kết xuất đã sắp xếp khi INSERTION-SORT hoàn tất.

```
INSERTION-SORT(A)
1 for j ← 2 to length[A]
2    do key ← A[j]
3       ▷ chèn A[j] vào chuỗi có sắp xếp A[1..j – 1].
4       i ← j – 1
5       while i > 0 và A[i] > key
6           do A[i + 1] ← A[i]
7              i ← i – 1
8       A[i + 1] ← key
```

Hình 1.2 nêu cách làm việc của thuật toán này với A = <5, 2, 4, 6, 1, 3>. Chỉ số j nêu rõ "lá bài hiện hành" đang được chèn vào tay. Các thành phần mảng $A[1..j - 1]$ tạo thành tay đang được sắp xếp, và các thành phần $A[j + 1..n]$ tương ứng với đống lá bài vẫn còn trên bàn. Chỉ số j dời từ trái sang phải qua mảng. Với mỗi lần lặp của vòng lặp **for** "phía ngoài", thành phần $A[j]$ được lấy ra khỏi mảng (dòng 2). Sau đó, bắt đầu tại vị trí j - 1, các thành phần liên tiếp được dời về bên phải một vị trí cho đến khi tìm thấy vị trí đúng đắn cho $A[j]$ (các dòng 4-7), tại điểm mà nó được chèn (dòng 8).



Hình 1.2 Phép toán INSERTION-SORT trên mảng A = <5, 2, 4, 6, 1, 3)>. Vị trí của chỉ số j được nêu rõ bởi một vòng tròn.

Các quy ước mã giả

Tập sách này sử dụng các quy ước dưới đây trong mã giả.

1. Thụt đầu dòng để nêu rõ cấu trúc khối. Ví dụ, thân của vòng lặp

for bắt đầu trên dòng 1 bao gồm các dòng 2-8, và thân của vòng lặp **while** bắt đầu trên dòng 5 chứa các dòng 6-7 nhưng không có dòng 8. Kiểu thụt dòng của chúng ta cũng áp dụng cho các câu lệnh **if-then-else**. Dùng kiểu thụt dòng thay vì các dấu chỉ quy ước của cấu trúc khối, như các câu lệnh **begin** và **end**, sẽ tránh được sự bừa bộn trong khi vẫn bảo toàn, thậm chí còn tăng cường, tính minh bạch[1].

2. Các kiến tạo vòng lặp **while, for,** và **repeat** và các kiến tạo điều kiện **if, then,** và **else** cũng được diễn dịch giống như trong Pascal.

3. Ký hiệu "▷" nêu rõ phần còn lại của dòng là một chú giải.

4. Kiểu nhiều phép gán theo dạng $i \leftarrow j \leftarrow e$ sẽ gán cho cả hai biến i và j giá trị của biểu thức e; nó sẽ được xem như tương đương với phép gán $j \leftarrow e$ theo sau là phép gán $i \leftarrow j$.

5. Các biến (như i, j, và *key*) là cục bộ đối với thủ tục đã cho. Ta không dùng các biến toàn cục mà không khai báo tường minh.

6. Các thành phần mảng được truy cập bằng cách đặc tả tên mảng theo sau là chỉ số trong các dấu ngoặc vuông. Ví dụ, $A[i]$ nêu rõ thành phần thứ i của mảng A. Ký hiệu ".." được dùng để nêu rõ một miền các giá trị trong một mảng. Do đó, $A[1..j]$ nêu rõ mảng con A bao gồm các thành phần $A[1], A[2], ..., A[j]$.

7. Dữ liệu phức hợp thường được tổ chức thành ***các đối tượng*** [objects], bao hàm ***các thuộc tính*** [attributes] hoặc ***các trường*** [fields]. Để truy cập một trường cụ thể, ta dùng tên trường theo sau là tên đối tượng của nó trong các dấu ngoặc vuông. Ví dụ, ta xem một mảng như một đối tượng có thuộc tính *length* nêu rõ số lượng thành phần mà nó chứa. Để đặc tả số lượng thành phần trong một mảng A, ta viết $length[A]$. Tuy ta dùng các dấu ngoặc vuông cho cả trường hợp chỉ số mảng lẫn các thuộc tính đối tượng, song nó vẫn rõ ý theo từng ngữ cảnh diễn dịch.

Một biến biểu thị cho một mảng hay đối tượng sẽ được xử lý như một biến trỏ [pointer] đến dữ liệu biểu thị cho mảng hay đối tượng đó. Với tất cả trường f của một đối tượng x, việc ấn định $y \leftarrow x$ sẽ khiến $f[y] = f[x]$. Vả lại, nếu giờ đây ta ấn định $f[x] \leftarrow 3$, thì sau đó không những là $f[x] = 3$, mà còn là $f[y] = 3$. Nói cách khác, x và y trỏ đến ("là") cùng đối tượng sau phép gán $y \leftarrow x$.

Đôi lúc, một biến trỏ không tham chiếu một đối tượng nào cả. Trong trường hợp này, ta gán cho nó giá trị đặc biệt NIL.

8. Các tham số được chuyển cho thủ tục ***theo giá trị:*** thủ tục được gọi

---

1 : Trong các ngôn ngữ lập trình thực thụ, ta không nên sử dụng một mình tính năng thụt dòng để nêu rõ cấu trúc khối, bởi các cấp thụt dòng khó xác định khi mã được tách thành nhiều trang.

sẽ nhận bản sao các tham số riêng của nó, và nếu nó gán một giá trị cho một tham số, thường trình gọi [calling routine] sẽ *không* thấy sự thay đổi đó. Khi các đối tượng được chuyển, biến trỏ đến dữ liệu biểu thị cho đối tượng đó sẽ được chép, song các trường của đối tượng thì không. Ví dụ, nếu x là một tham số của một thủ tục được gọi [called procedure], thủ tục gọi sẽ không thấy phép gán $x \leftarrow y$ trong thủ tục được gọi. Tuy nhiên, phép gán $f[x] \leftarrow 3$ lại lộ diện.

Bài tập

1.1-1

Dùng Hình 1.2 làm mẫu, hãy minh họa phép toán của thủ tục INSERTION-SORT trên mảng A = <31, 41, 59, 26, 41, 58>.

1.1-2

Viết lại thủ tục INSERTION-SORT để sắp xếp theo thứ tự không tăng [nonincreasing] thay vì không giảm [nondecreasing].

1.1-3

Cho ***bài toán tìm kiếm:***

Đầu vào: Một dãy n con số $A = (a_1, a_2, \ldots, a_n)$ và một giá trị v.

Kết xuất: Một chỉ số i sao cho $v = A[i]$ hoặc giá trị đặc biệt NIL nếu v không xuất hiện trong A.

Viết mã giả cho kỹ thuật ***tìm kiếm tuyến tính,*** quét qua dãy, tìm v.

1.1-4

Xét bài toán cộng hai số nguyên nhị phân n-bit, lưu trữ trong hai mảng A và B n-thành phần. Tổng hai số nguyên sẽ được lưu trữ theo dạng nhị phân trong một mảng C $(n+1)$-thành phần. Phát biểu bài toán theo hình thức và viết mã giả để cộng hai số nguyên.

1.2 Phân tích các thuật toán

Phân tích một thuật toán thường hàm ý tiên liệu các tài nguyên mà thuật toán yêu cầu. Thỉnh thoảng, các tài nguyên như bộ nhớ, băng thông, hoặc các cổng lôgic là những yếu tố được quan tâm hàng đầu, song đa phần chính thời gian tính toán mới là yếu tố mà ta cần đo lường. Nói chung, nhờ phân tích vài thuật toán ứng tuyển của một bài toán, ta có thể dễ dàng nhận ra thuật toán nào là hiệu quả nhất. Kiểu phân tích như vậy có thể nêu rõ nhiều ứng viên tồn tại, song một vài thuật toán kém hơn thường bị loại trong khi tiến hành.

Để có thể phân tích một thuật toán, ta phải áp dụng một mô hình

công nghệ thực thi, kể cả mô hình cho các tài nguyên của công nghệ đó và các chi phí của chúng. Đa phần tập sách này mặc nhận sử dụng mô hình điện toán ***RAM*** (*random-access machine = máy truy cập ngẫu nhiên*), một bộ xử lý chung, làm công nghệ thực thi và ngầm hiểu rằng các thuật toán sẽ được thực thi dưới dạng các chương trình máy tính. Trong mô hình RAM, các chỉ lệnh được thi hành lần lượt, mà không có các phép toán đồng thời. Tuy nhiên, trong các chương sau, ta sẽ có dịp nghiên cứu các mô hình của các máy tính song song và phần cứng số hóa.

Quá trình phân tích luôn là một thách thức, thậm chí với một thuật toán đơn giản. Các công cụ toán học cần thiết có thể gồm cả toán học tổ hợp trừu tượng, lý thuyết xác suất căn bản, kỹ năng về đại số, và khả năng định danh các số hạng quan trọng nhất trong một công thức. Do cách ứng xử của một thuật toán có thể khác nhau đối với từng đầu vào khả dĩ, nên ta cần có một biện pháp để tóm lược cách ứng xử thành các công thức đơn giản và dễ hiểu.

Cho dù thông thường chỉ lựa mô hình một máy để phân tích một thuật toán nào đó, song ta vẫn phải đối mặt với nhiều chọn lựa khi quyết định cách diễn tả tiến trình phân tích. Một mục tiêu tức thời đó là tìm một biện pháp diễn tả đơn giản để viết và điều tác [manipulate], nêu các đặc tính quan trọng của các yêu cầu tài nguyên của một thuật toán, và hủy bỏ các chi tiết dài dòng.

Phân tích kỹ thuật sắp xếp chèn

Thời gian kéo dài của thủ tục INSERTION-SORT thường tùy thuộc vào đầu vào: tiến trình sắp xếp một ngàn con số sẽ lâu hơn tiến trình sắp xếp ba con số. Vả lại, INSERTION-SORT có thể sử dụng các thời lượng khác nhau để sắp xếp hai dãy đầu vào có kích cỡ giống nhau, tùy thuộc vào mức độ sắp xếp sẵn của chúng. Nói chung, thời gian thực hiện của một thuật toán thường tăng theo kích cỡ đầu vào, do đó theo truyền thống, ta thường mô tả thời gian thực hiện của một chương trình như một hàm kích cỡ đầu vào của chương trình đó. Để thực hiện, ta cần định nghĩa các thuật ngữ "thời gian thực hiện" [running time] và "kích cỡ đầu vào" [size of input] cẩn thận hơn.

Ý niệm thích hợp nhất của ***kích cỡ đầu vào*** thường tùy thuộc vào bài toán đang nghiên cứu. Với nhiều bài toán, như sắp xếp hoặc tính toán các phép biến đổi Fourier, số đo tự nhiên nhất đó là *số lượng các mục*

trong đầu vào—ví dụ, kích cỡ mảng *n* để sắp xếp. Với nhiều bài toán khác, như nhân hai số nguyên, số đo tốt nhất của kích cỡ đầu vào lại là *tổng số bit* cần thiết để biểu thị đầu vào theo hệ ký hiệu nhị phân bình thường. Đôi lúc, việc mô tả kích cỡ đầu vào bằng hai con số thay vì một lại tỏ ra thích hợp hơn. Ví dụ, nếu đầu vào cho một thuật toán là một đồ thị, kích cỡ đầu vào có thể được mô tả bởi các số đỉnh [vertices] và các cạnh trong đồ thị. Ta sẽ nêu rõ kiểu đo kích cỡ đầu vào sẽ được dùng với từng bài toán mà ta nghiên cứu.

Thời gian thực hiện [running time] của một thuật toán trên một đầu vào cụ thể chính là số lượng phép toán nguyên tố [primitive operations] hoặc "các bước" [steps] được thi hành. Sẽ tiện dụng hơn nếu ta định nghĩa khái niệm "bước" để nó càng độc lập máy càng tốt. Trước mắt, hãy chấp nhận quan điểm sau. Cần có một thời lượng bất biến để thi hành từng dòng mã giả của chúng ta. Dòng này có thể mất một thời lượng khác với dòng kia, song ta mặc nhận rằng từng đợt thi hành dòng thứ i sẽ mất một thời gian c_i, ở đó c_i là một hằng. Quan điểm này phù hợp với mô hình RAM, và nó cũng phản ánh cách thực thi mã giả trên hầu hết các máy tính hiện nay[2].

Trong đoạn mô tả dưới đây, cách diễn tả của chúng ta về thời gian thực hiện của INSERTION-SORT sẽ tiến hóa từ một công thức hỗn độn sử dụng tất cả mọi hao phí câu lệnh c_i thành một hệ ký hiệu đơn giản hơn nhiều, dễ dàng điều tác và súc tích hơn. Hệ ký hiệu đơn giản này cũng sẽ giúp ta dễ dàng xác định xem thuật toán này có hiệu quả hơn thuật toán kia hay không.

Để bắt đầu, ta trình bày thủ tục INSERTION-SORT bằng các mức "hao phí" thời gian của từng câu lệnh và số lần thi hành từng câu lệnh. Với mỗi $j = 2,3,\ldots,n$, ở đó $n = length[A]$, ta giả sử t_j là số lần thi hành đợt trắc nghiệm vòng lặp **while** trong dòng 5 theo giá trị j đó. Ta mặc nhận rằng các chú giải không phải là các câu lệnh thi hành, và do đó không mất thời gian.

---

[2] Ở đây có vài điểm tinh tế. Các bước tính toán mà ta đặc tả bằng tiếng Anh thường là các biến thể của một thủ tục yêu cầu không chỉ một thời lượng bất biến. Ví dụ, ở phần sau trong cuốn sách này, ta có thể nói "sort the points by x-coordinate" [sắp xếp các điểm theo tọa độ x], mà như sẽ thấy, sẽ vận dụng nhiều hơn một thời lượng bất biến. Ngoài ra cũng lưu ý, một câu lệnh gọi một chương trình con sẽ mất một thời lượng bất biến, tuy rằng một khi được triệu gọi, chương trình con có thể mất nhiều thời gian hơn. Nghĩa là, ta tách riêng tiến trình ***gọi*** chương trình con—chuyển các tham số cho nó, vân vân—với tiến trình ***thi hành*** chương trình con ***đó***.

	INSERTION-SORT(*A*)	*costs*	*times*
1	**for** *j* ← 2 **to** *length* [*A*]	c_1	n
2	**do** *key* ← *A*[*j*]	c_2	$n-1$
3	▷ chèn *A*[*j*] vào chuỗi có sắp xếp		
	▷ sequence *A*[1..*j* − 1].	0	$n-1$
4	*i* ← *j* − 1	c_4	$n-1$
5	**while** *i* > 0 and *A*[*i*] > *key*	c_5	$\sum_{j=2}^{n} t_j$
6	**do** *A* [*i* + 1] ← *A*[*i*]	c_6	$\sum_{j=2}^{n}(t_j-1)$
7	*i* ← *i* − 1	c_7	$\sum_{j=2}^{n}(t_j-1)$
8	*A*[*i* + 1] ← *key*	c_8	$n-1$

Thời gian thực hiện của thuật toán là tổng các thời gian thực hiện của từng câu lệnh được thi hành; một câu lệnh trải qua các bước c_i để thi hành và được thi hành n lần sẽ đóng góp $c_i n$ vào tổng thời gian thực hiện[3]. Để tính $T(n)$, thời gian thực hiện của INSERTION-SORT, ta tổng cộng các tích của các cột *costs* và *times*, thành

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

Thậm chí với các đầu vào có một kích cỡ nhất định, thời gian thực hiện của một thuật toán có thể tùy thuộc vào việc cho đầu vào *nào* có kích cỡ đó. Ví dụ, trong INSERTION-SORT, trường hợp tốt nhất xảy ra khi mảng đã được sắp xếp sẵn. Với mỗi $j = 2, 3, ..., n$, ta thấy rằng $A[i] \le key$ trong dòng 5 khi i có giá trị ban đầu là $j - 1$. Như vậy $t_j = 1$ với $j = 2, 3,..., n$, và thời gian thực hiện trong trường hợp tốt nhất là:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8). \end{aligned}$$

Thời gian thực hiện này có thể được diễn tả là: $an + b$ với các hằng a và b, tùy thuộc vào hao phí câu lệnh c_i; do đó nó là một ***hàm tuyến tính*** của n.

Nếu mảng được sắp xếp theo thứ tự đảo ngược—nghĩa là, theo thứ tự giảm—trường hợp xấu nhất sẽ xảy ra. Ta phải so sánh mỗi thành phần $A[j]$ với mỗi thành phần trong nguyên cả mảng con đã sắp xếp $A[1..j-1]$, và như vậy $t_j = j$ với $j = 2, 3,..., n$. Lưu ý rằng

---

[3] Đặc tính này không nhất thiết áp dụng cho một tài nguyên như bộ nhớ. Một câu lệnh tham chiếu *m* từ [words] của bộ nhớ và được thi hành *n* lần không nhất thiết tiêu thụ tổng cộng *mn* từ của bộ nhớ.

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

và

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

(Chương 3 sẽ đề cập lại các phép tổng này), ta thấy rằng trong trường hợp xấu nhất, thời gian thực hiện của INSERTION-SORT là

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right)$$
$$+ c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2} - 1\right) + c_8(n-1)$$
$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} + c_8\right)n$$
$$- (c_2 + c_4 + c_5 + c_8)$$

Thời gian thực hiện trường hợp xấu nhất này có thể được diễn tả là $an^2 + bn + c$ với các hằng *a, b,* và *c,* mà một lần nữa tùy thuộc vào hao phí câu lệnh c_i; do đó nó là một ***hàm bậc hai*** của *n.*

Thông thường, như trong trường hợp sắp xếp chèn, thời gian thực hiện của một thuật toán được cố định theo một đầu vào đã cho, mặc dù trong các chương sau, ta sẽ gặp các thuật toán "ngẫu nhiên hóa" [randomized] đáng quan tâm có cách ứng xử có thể thay đổi thậm chí với cả đầu vào cố định.

Phân tích trường hợp xấu nhất và trường hợp trung bình

Trong kỹ thuật phân tích sắp xếp chèn trên đây, ta đã xem xét cả ca tốt nhất, ở đó mảng đầu vào đã được sắp xếp sẵn, lẫn trường hợp xấu nhất, ở đó mảng đầu vào được sắp xếp đảo ngược. Tuy nhiên, với phần còn lại của cuốn sách, ta thường tập trung vào việc chỉ tìm thời gian thực hiện trường hợp xấu nhất; nghĩa là, thời gian thực hiện dài nhất của một đầu vào *bất kỳ* có kích cỡ *n.* Sở dĩ như vậy là vì ba lý do khả dĩ sau đây.

- Thời gian thực hiện trường hợp xấu nhất của một thuật toán là một cận trên [upper bound] đối với thời gian thực hiện của một đầu vào bất kỳ. Biết rằng nó bảo đảm thuật toán sẽ không bao giờ kéo dài hơn nữa. Ta không cần phải suy đoán này nọ về thời gian thực hiện và hy vọng nó không trở nên tệ hại hơn.

- Với vài thuật toán, trường hợp xấu nhất xảy ra khá thường xuyên. Ví dụ, trong khi tìm kiếm một mẩu thông tin cụ thể trong một cơ sở dữ liệu, trường hợp xấu nhất của thuật toán tìm kiếm thường xảy ra khi

thông tin đó không hiện diện trong cơ sở dữ liệu. Trong vài ứng dụng tìm kiếm, ta thường gặp các đợt tìm kiếm thông tin vắng mặt.

Thông thường, "trường hợp trung bình" [average case] cũng tệ hại tương tự như trường hợp xấu nhất. Giả sử, ta ngẫu nhiên chọn n con số và áp dụng kỹ thuật sắp xếp chèn. Phải mất bao lâu để xác định vị trí chèn thành phần $A[j]$ trong mảng con $A[1..j-1]$? Tính trung bình, phân nửa các thành phần trong $A[1..j-1]$ là nhỏ hơn $A[j]$, và phân nửa các thành phần là lớn hơn. Như vậy, tính trung bình, ta kiểm tra phân nửa mảng con $A[1..j-1]$, do đó $t_j = j/2$. Nếu tính ra thời gian thực hiện của trường hợp trung bình kết quả, thì hóa ra đó là một hàm bậc hai của kích cỡ đầu vào, hệt như thời gian thực hiện trường hợp xấu nhất.

Trong vài trường hợp cụ thể, ta sẽ quan tâm đến thời gian thực hiện ***dự trù*** [expected] hoặc ***trường hợp trung bình*** [average case] của một thuật toán. Tuy nhiên, một bài toán có tiến hành phân tích trường hợp trung bình đó là: khi không nắm rõ nội dung tạo thành một đầu vào "trung bình" cho một bài toán cụ thể. Thông thường, ta sẽ mặc nhận rằng tất cả đầu vào có một kích cỡ nhất định có thể xảy ra như nhau. Trong thực tế, giả thiết này có thể bị vi phạm, song đôi lúc một thuật toán ngẫu nhiên hóa có thể buộc phải áp dụng nó.

Cấp tăng trưởng

Ta đã dùng vài kiểu trừu tượng giản lược để tạo thuận lợi cho tiến trình phân tích thủ tục INSERTION-SORT. Trước tiên, ta đã bỏ qua mức hao phí [cost] thực tế của từng câu lệnh, dùng các hằng c_i để biểu thị các hao phí này. Sau đó, ta nhận thấy thậm chí các hằng này còn cho ta nhiều chi tiết hơn mức cần thiết thực tế: thời gian thực hiện ca xấu nhất là $an^2 + bn + c$ với một số hằng a, b, và c tùy thuộc vào các hao phí câu lệnh c_i. Như vậy, ta đã bỏ qua không những các hao phí câu lệnh thực tế, mà cả các hao phí trừu trượng [abstract costs] c_i.

Giờ đây, ta sẽ xem xét một kiểu trừu tượng giản lược khác. Đó là ***tốc độ tăng trưởng*** [rate of growth], hoặc ***cấp tăng trưởng***, của thời gian thực hiện mà ta thực sự quan tâm. Vì vậy, ta chỉ xem xét số hạng đầu của một công thức (ví dụ, an^2), bởi các số hạng cấp thấp tương đối không quan trọng với n lớn. Ta cũng bỏ qua hệ số bất biến của số hạng đầu, bởi các thừa số bất biến ít quan trọng hơn tốc độ tăng trưởng trong khi xác định hiệu năng tính toán của các đầu vào lớn. Do đó, ta viết, chẳng hạn, kỹ thuật sắp xếp chèn có một thời gian thực hiện trường hợp xấu nhất là $\Theta(n^2)$ (đọc là "theta của n-bình phương"). Trong chương này, ta đơn giản dùng ký hiệu Θ; nó sẽ được định nghĩa một cách chính

xác trong Chương 2.

Một thuật toán thường được xem là hiệu quả hơn so với một thuật toán khác khi thời gian thực hiện trường hợp xấu nhất của nó có một cấp tăng trưởng thấp hơn. Kiểu đánh giá này có thể gặp lỗi đối với các đầu vào nhỏ, song với các đầu vào đủ lớn, thì trong trường hợp xấu nhất một thuật toán như $\Theta(n^2)$ chẳng hạn sẽ chạy nhanh hơn so với một thuật toán $\Theta(n^3)$.

Bài tập

1.2-1

Xem xét tiến trình sắp xếp n con số lưu trữ trong mảng A bằng cách trước tiên tìm thành phần nhỏ nhất của A và đặt nó trong mục đầu tiên của một mảng B khác. Sau đó tìm thành phần nhỏ nhất thứ hai của A và đặt nó trong mục thứ hai của B. Tiếp tục như vậy với n thành phần của A. Viết mã giả của thuật toán này, có tên là ***sắp xếp chọn lọc***. Nêu các thời gian thực hiện trường hợp tốt nhất và xấu nhất của kỹ thuật sắp xếp chọn lọc theo hệ ký hiệu .

1.2-2

Xem xét lại kiểu tìm kiếm tuyến tính (xem Bài tập 1.1-3). Giả định thành phần đang tìm kiếm có khả năng như nhau là một thành phần bất kỳ trong mảng, ta cần kiểm tra trung bình bao nhiêu thành phần của dãy đầu vào? Trong trường hợp xấu nhất thì sao? Nêu các thời gian thực hiện trường hợp xấu nhất và trung bình của kỹ thuật tìm kiếm tuyến tính theo hệ ký hiệu Θ Chứng minh các nghiệm.

1.2-3

Xem xét bài toán xác định xem một dãy tùy ý n con số $\langle x_1, x_2, \ldots, x_n \rangle$ có chứa các lần xuất hiện lặp lại của một con số nào đó hay không. Dẫn chứng điều này có thể thực hiện theo thời gian $\Theta(n \lg n)$, ở đó $\lg n$ là $\log_2 n$.

1.2-4

Xem xét bài toán đánh giá một đa thức tại một điểm. Cho n hệ số a_0, $a_1, \ldots, a_{n-1}$ và một số thực x, ta muốn tính toán $\sum_{i=0}^{n-1} a_i x^i$. Mô tả một thuật toán dễ hiểu $\Theta(n^2)$ cho bài toán này. Mô tả một thuật toán $\Theta(n)$ - thời gian sử dụng phương pháp dưới đây (có tên quy tắc Horner) để viết lại đa thức:

$$\sum_{i=0}^{n-1} a_i x^i = (\ldots(a_{n-1}x + a_{n-2})x + \ldots + a_1)x + a_0.$$

1.2-5

Biểu diễn hàm $n^3/1000 - 100n^2 - 100n + 3$ theo hệ ký hiệu Θ.

1.2-6

Làm sao để sửa đổi hầu hết mọi thuật toán để có một thời gian thực hiện tốt trong trường hợp tốt nhất?

1.3 Thiết kế các thuật toán

Có nhiều cách để thiết kế các thuật toán. Phương pháp sắp xếp chèn sử dụng cách tiếp cận ***gia số*** [incremental]: khi sắp xếp mảng con $A[1..j - 1]$, ta chèn một thành phần $A[j]$ vào đúng chỗ của nó, cho ra mảng con đã sắp xếp $A[1..j]$.

Trong đoạn này, ta xem xét một cách tiếp cận thiết kế khác, có tên "chia để trị" ["divide-and-conquer"]. Ta sẽ dùng kỹ thuật chia để trị để thiết kế một thuật toán sắp xếp có thời gian thực hiện trường hợp xấu nhất nhỏ hơn nhiều so với kiểu sắp xếp chèn. Các thuật toán chia để trị có một ưu điểm đó là các thời gian thực hiện của chúng thường dễ xác định bằng các kỹ thuật mô tả trong Chương 4.

1.3.1 Cách tiếp cận chia để trị

Có nhiều thuật toán hữu ích theo cấu trúc ***đệ quy*** : để giải quyết một bài toán nhất định, chúng tự gọi theo đệ quy một hay nhiều lần để đối phó với các bài toán con có liên quan mật thiết. Các thuật toán này thường theo cách tiếp cận ***chia để trị***: chúng tách nhỏ bài toán thành vài bài toán con tương tự như bài toán ban đầu song có kích cỡ nhỏ hơn, giải quyết các bài toán con một cách đệ quy, rồi tổ hợp các nghiệm này để tạo một nghiệm cho bài toán ban đầu.

Kiểu mẫu chia để trị liên quan đến ba bước tại mỗi cấp của đệ quy:

Chia bài toán thành một số bài toán con.

"Trị" các bài toán con bằng cách giải quyết chúng một cách đệ quy. Tuy nhiên, nếu các bài toán con đủ nhỏ, ta chỉ việc giải quyết các bài toán con theo cách đơn giản.

Tổ hợp các nghiệm của các bài toán con thành nghiệm cho bài toán ban đầu.

Thuật toán ***sắp xếp trộn*** [merge sort] theo sát kiểu mẫu chia để trị. Theo trực giác, nó hoạt động như sau.

Chia: Chia dãy n-thành phần sẽ được sắp xếp thành hai dãy con, mỗi dãy có $n/2$ thành phần.

Trị: Dùng kỹ thuật sắp xếp trộn để sắp xếp hai dãy con theo đệ quy.

Tổ hợp: Hợp nhất hai dãy con đã sắp xếp để cho ra đáp án đã sắp xếp.

Lưu ý, đệ quy sẽ "hóa ra công cốc" khi dãy sắp xếp có chiều dài là 1, trong trường hợp đó không có gì được thực hiện, bởi mọi dãy có chiều dài 1 đã theo một thứ tự sắp xếp sẵn.

Tác vụ chính của thuật toán sắp xếp trộn đó là tiến trình trộn hai dãy đã sắp xếp trong bước "tổ hợp". Để thực hiện tiến trình trộn, ta dùng một thủ tục phụ trợ MERGE(A, p, q, r), ở đó A là một mảng và p, q, và r là các chỉ mục đánh số các thành phần của mảng sao cho $p \leq q < r$. Thủ tục mặc nhận các mảng con $A[p..q]$ và $A[q+1..r]$ nằm theo thứ tự sắp xếp. Nó ***trộn*** [merges] chúng để tạo thành chỉ một mảng con sắp xếp thay thế mảng con hiện hành $A[p..r]$.

Mặc dù để dành mã giả làm bài tập (xem Bài tập 1.3-2), song ta cũng dễ dàng hình dung một thủ tục MERGE bỏ ra một thời gian $\Theta(n)$, ở đó $n = r - p + 1$ là số lượng thành phần đang được trộn. Trở về chủ đề chơi bài, giả sử ta có hai đống lá bài tung ngửa trên bàn. Mỗi đống được sắp xếp, với lá bài nhỏ nhất nằm trên cùng. Ta muốn trộn hai đống thành chỉ một đống kết xuất đã sắp xếp, nằm tung sấp trên bàn. Bước căn bản bao gồm tiến trình chọn lá nhỏ hơn trong hai lá bài trên đầu của các đống tung ngửa, gỡ bỏ nó ra khỏi đống đó (bày ra một lá bài mới trên đầu), và đặt lá bài này tung sấp lên trên đống kết xuất. Ta lặp lại bước này cho đến khi một đống đầu vào trống rỗng, vào lúc đó ta chỉ việc lấy đống đầu vào còn lại và đặt nó tung sấp lên trên đống kết xuất. Về mặt tính toán, mỗi bước căn bản bỏ ra một thời gian bất biến, bởi ta chỉ đang kiểm tra hai lá bài trên cùng. Do ta thực hiện tối đa n bước căn bản, nên tiến trình trộn chiếm $\Theta(n)$ thời gian.

Giờ đây, có thể dùng thủ tục MERGE như một chương trình con trong thuật toán sắp xếp trộn. Thủ tục MERGE-SORT (A, p, r) sắp xếp các thành phần trong mảng con $A[p..r]$. Nếu $p \geq r$, mảng con có tối đa một thành phần và do đó đã được sắp xếp sẵn. Bằng không, bước chia đơn giản tính toán một chỉ số q phân hoạch $A[p..r]$ thành hai mảng con: $A[p..q]$, chứa $\lceil n/2 \rceil$ thành phần, và $A[q+1..r]$, chứa $\lfloor n/2 \rfloor$ thành phần[4].

MERGE-SORT (A, p, r)

1 **if** $p < r$

2 **then** $q \leftarrow \lfloor (p+r)/2 \rfloor$

---

4 Dạng thức $\lceil x \rceil$ thể hiện số nguyên thấp nhất lớn hơn hoặc bằng x, và $\lfloor x \rfloor$ thể hiện số nguyên lớn nhất nhỏ hơn hoặc bằng x. Các ký hiệu này sẽ được định nghĩa trong Chương 2.

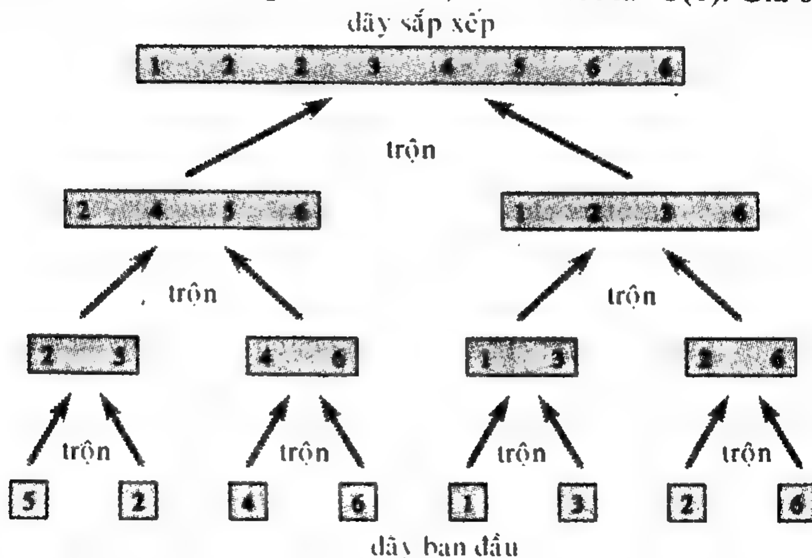
```
MERGE-SORT(A,p,q)
MERGE-SORT(A, q + 1, r)
MERGE(A, p, q, r)
```

Để sắp xếp nguyên cả dãy $A = <A[1], A[2], ..., A[n]>$, ta gọi MERGE-SORT($A$, 1, *length*[$A$]), ở đó một lần nữa *length*[A] = n. Nếu xem xét phép toán của thủ tục từ dưới lên khi n là một lũy thừa của hai, thuật toán bao gồm tiến trình trộn các cặp dãy 1-mục để tạo thành các dãy sắp xếp có chiều dài 2, tiến trình trộn các cặp dãy có chiều dài 2 để tạo thành các dãy sắp xếp có chiều dài 4, và vân vân, cho đến khi hai dãy có chiều dài $n/2$ được trộn để tạo thành dãy sắp xếp chung cuộc có chiều dài n. Hình 1.3 minh họa tiến trình này.

1.3.2 Phân tích các thuật toán chia để trị

Khi một thuật toán chứa một lệnh gọi đệ quy lên chính nó, thời gian thực hiện của nó thường được mô tả bởi một ***phương trình truy toán*** hoặc ***phép truy toán***, mô tả thời gian thực hiện chung trên một bài toán có kích cỡ n theo dạng thời gian thực hiện trên các đầu vào nhỏ hơn. Như vậy, có thể dùng các công cụ toán học để giải quyết phép truy toán và cung cấp các giới hạn về khả năng thực hiện của thuật toán.

Một phép truy toán của thời gian thực hiện trong một thuật toán chia để trị thường dựa trên ba bước kiểu mẫu căn bản. Cũng như trước, giả sử $T(n)$ là thời gian thực hiện trên một bài toán có kích cỡ n. Nếu kích cỡ bài toán đủ nhỏ, giả sử $n \leq c$ với một hằng c nào đó, nghiệm đơn giản sẽ bỏ ra một thời gian bất biến, mà ta viết là $\Theta(1)$. Giả sử ta chia



Hình 1.3 Phép toán sắp xếp trộn trên mảng $A = <5, 2, 4, 6, 1, 3, 2, 6>$. Chiều dài của các dãy sắp xếp đang được trộn sẽ gia tăng khi thuật toán tiến dần từ dưới lên trên.

bài toán thành a bài toán con, mỗi bài có $1/b$ kích cỡ so với bài toán ban đầu. Nếu lấy $D(n)$ thời gian để chia bài toán thành các bài toán con và $C(n)$ thời gian để tổ hợp các nghiệm của các bài toán con thành nghiệm cho bài toán ban đầu, ta sẽ có phép truy toán

$$T(n) = \begin{cases} \Theta(1) & \text{nếu } n \le c, \\ aT(n/b) + D(n) = C(n) & \text{bằng không} \end{cases}$$

Trong Chương 4, ta sẽ xem cách giải quyết các phép truy toán chung theo dạng này.

Phân tích kỹ thuật sắp xếp trộn

Tuy mã giả của MERGE-SORT làm việc đúng đắn khi số lượng thành phần không đều, song ta có thể đơn giản hóa tiến trình phân tích gốc-phép truy toán nếu như mặc nhận kích cỡ bài toán ban đầu là một lũy thừa của hai. Như vậy, mỗi bước chia cho ra hai dãy con có kích cỡ chính xác là $n/2$. Trong Chương 4, ta sẽ thấy giả thiết này không ảnh hưởng đến cấp tăng trưởng của giải pháp đối với phép truy toán.

Dưới đây là cách suy luận để xác lập phép truy toán cho $T(n)$, là thời gian thực hiện trường hợp xấu nhất của phương pháp sắp xếp trộn trên n con số. Tiến trình sắp xếp trộn trên chỉ một thành phần sẽ bỏ ra một thời gian bất biến. Khi có $n > 1$ thành phần, ta tách nhỏ thời gian thực hiện như sau.

Chia: Bước chia chỉ tính toán điểm giữa của mảng con, bỏ ra một thời gian bất biến. Do đó, $D(n) = \Theta(1)$.

Trị: Ta giải quyết đệ quy hai bài toán con, mỗi bài có kích cỡ $n/2$, đóng góp $2T(n/2)$ vào thời gian thực hiện.

Tổ hợp: Ta đã lưu ý thủ tục MERGE trên một mảng con n-thành phần bỏ ra một thời gian $\Theta(n)$, do đó $C(n) = \Theta(n)$.

Khi cộng các hàm $D(n)$ và $C(n)$ của đợt phân tích sắp xếp trộn, ta đang cộng một hàm là $\Theta(n)$ và một hàm là $\Theta(1)$. Tổng này là một hàm tuyến tính của n, tức, $\Theta(n)$. Việc cộng nó với số hạng $2T(n/2)$ của bước "trị" [conquer] sẽ cho ra phép truy toán cho thời gian thực hiện trường hợp xấu nhất $T(n)$ của kỹ thuật sắp xếp trộn:

$$T(n) = \begin{cases} \Theta(1) & \text{nếu } n = 1, \\ 2T(n/2) + \Theta(n) & \text{nếu } n > 1. \end{cases}$$

Trong Chương 4, ta sẽ thấy $T(n)$ là $\Theta(n \lg n)$, ở đó $\lg n$ thay cho $\log_2 n$. Nếu các đầu vào đủ lớn, kỹ thuật sắp xếp trộn, với thời gian thực hiện $\Theta(n \lg n)$ của nó, tỏ ra trội vượt so với kỹ thuật sắp xếp chèn, có thời gian thực hiện là $\Theta(n^2)$ trong trường hợp xấu nhất.

Bài tập

1.3-1

Dùng Hình 1.3 làm mẫu, minh họa phép toán sắp xếp trộn trên mảng $A = <3, 41, 52, 26, 38, 57, 9, 49>$.

1.3-2

Viết mã giả của MERGE(A, p, q, r).

1.3-3

Dùng phép quy nạp toán học để nêu rõ nghiệm của phép truy toán

$$T(n) = \begin{cases} 2 & \text{nếu } n = 2, \\ 2T(n/2) + n & \text{nếu } n > 2^k, k > 1. \end{cases}$$

là $T(n) = n\lg n$.

1.3-4

Sắp xếp chèn có thể được diễn tả dưới dạng một thủ tục đệ quy như sau. Để sắp xếp $A[1..n]$, ta sắp xếp theo đệ quy $A[1..n - 1]$ rồi chèn $A[n]$ vào mảng đã sắp xếp $A[1..n - 1]$. Viết một phép truy toán cho thời gian thực hiện của phiên bản đệ quy này của kỹ thuật sắp xếp chèn.

1.3-5

Quay lại bài toán tìm kiếm (xem Bài tập 1.1-3), nhận thấy nếu dãy A được sắp xếp, ta có thể kiểm tra điểm giữa của dãy theo v và loại bỏ phân nửa dãy không cần xét thêm. ***Tìm nhị phân*** [binary search] là một thuật toán lặp lại thủ tục này, mỗi lần chia đôi kích cỡ phần còn lại của dãy. Viết mã giả, lặp lại hoặc đệ quy, cho kỹ thuật tìm nhị phân. Chứng tỏ thời gian thực hiện trường hợp xấu nhất của kỹ thuật tìm nhị phân là $\Theta(\lg n)$.

1.3-6

Nhận thấy vòng lặp **while** của các dòng 5-7 trong thủ tục INSERTION-SORT ở Đoạn 1.1 sử dụng một đợt tìm kiếm tuyến tính để quét (lùi) qua mảng con sắp xếp $A[1 .. j - 1]$. Có thể dùng kỹ thuật tìm nhị phân (xem Bài tập 1.3- 5) để cải thiện thời gian thực hiện chung trong trường hợp xấu nhất của kỹ thuật sắp xếp chèn theo $\Theta(n \lg n)$ hay không?

1.3-7 *

Mô tả một thuật toán $\Theta(n \lg n)$-thời gian để, cho một tập hợp S gồm n số thực và một số thực x khác, xác định có tồn tại hay không hai thành phần trong S có tổng chính xác là x.

1.4 Tóm tắt

Một thuật toán tốt cũng giống như một con dao sắc—nó thực hiện

chính xác những gì cần làm với một nỗ lực tối thiểu. Dùng thuật toán sai để giải quyết một bài toán cũng giống như đang gắng dùng một tua-vít để cắt một miếng thịt nướng: tuy chung cuộc vẫn có thể có một kết quả tiêu hóa được, song bạn phải tốn nhiều nỗ lực hơn mức cần thiết, mà kết quả không mấy thẩm mỹ.

Các thuật toán được sáng chế để giải quyết cùng một bài toán thường khác biệt đáng kể về hiệu năng. Những khác biệt này có thể có ý nghĩa hơn nhiều so với sự khác biệt giữa một máy tính cá nhân và một siêu máy tính. Ví dụ, hãy để một siêu máy tính chạy phương pháp sắp xếp chèn đấu với một máy tính cá nhân nhỏ chạy phương pháp sắp xếp trộn. Chúng phải sắp xếp một mảng gồm một triệu con số. Giả sử siêu máy tính thi hành 100 triệu chỉ lệnh mỗi giây, trong khi máy tính cá nhân chỉ thi hành một triệu chỉ lệnh mỗi giây. Để phóng đại thêm sự khác biệt này, ta giả sử một lập trình viên lanh lợi nhất thế giới đã mã hóa kỹ thuật sắp xếp chèn bằng ngôn ngữ máy cho siêu máy tính, và mã kết quả đó yêu cầu $2n^2$ chỉ lệnh siêu máy tính để sắp xếp n con số. Mặt khác, kỹ thuật sắp xếp trộn được một lập trình viên trung bình viết mã cho máy tính cá nhân bằng một ngôn ngữ cấp cao với một trình biên dịch không mấy hiệu quả, và mã kết quả sử dụng $50n \lg n$ chỉ lệnh máy tính cá nhân. Để sắp xếp một triệu con số, siêu máy tính bỏ ra

$$\frac{2 \cdot (10^6)^2 \text{ chỉ lệnh}}{10^8 \text{ chỉ lệnh/giây}} = 20{,}000 \text{ giây} \approx 5.56 \text{ giờ},$$

trong khi máy tính cá nhân bỏ ra

$$\frac{50 \cdot 106 \lg 10^6 \text{ chỉ lệnh}}{10^6 \text{ chỉ lệnh/giây}} \approx 1{,}000 \text{ giây} \approx 16.67 \text{ phút}.$$

Nhờ dùng một thuật toán có thời gian thực hiện theo cấp tăng trưởng thấp hơn, thậm chí với một bộ biên dịch tồi, máy tính cá nhân vẫn chạy nhanh hơn gấp 20 lần so với siêu máy tính!

Ví dụ này cho thấy các thuật toán, cũng giống như phần cứng máy tính, là một ***công nghệ***. Tổng khả năng thực hiện của hệ thống tùy thuộc vào việc chọn các thuật toán hiệu quả chẳng khác gì với việc chọn phần cứng chạy nhanh. Giống như trong các công nghệ máy tính khác, các tiến bộ trong lĩnh vực thuật toán cũng đang diễn ra một cách nhanh chóng.

Bài tập

1.4-1

Giả sử ta đang so sánh các cách thực thi phương pháp sắp xếp chèn

và sắp xếp trộn trên cùng một máy. Với các đầu vào có kích cỡ n, tiến trình sắp xếp chèn chạy trong $8n^2$ bước, trong khi tiến trình sắp xếp trộn chạy trong $64n \lg n$ bước. Với các giá trị n nào, phương pháp sắp xếp chèn sẽ đánh bại phương pháp sắp xếp trộn? Làm sao viết lại mã giả sắp xếp trộn để nó thậm chí chạy nhanh hơn trên các đầu vào nhỏ?

1.4-2

Nêu giá trị n nhỏ nhất sao cho một thuật toán có thời gian thực hiện là $100n^2$ chạy nhanh hơn một thuật toán có thời gian thực hiện là 2^n trên cùng một máy?

Các bài toán

1-1 So sánh các thời gian thực hiện

Với mỗi hàm $f(n)$ và thời gian t trong bảng dưới đây, hãy xác định kích cỡ n lớn nhất của một bài toán mà ta có thể giải trong thời gian t, giả định thuật toán để giải quyết bài toán bỏ ra $f(n)$ microgiây.

	1 giây	1 phút	1 giờ	1 ngày	1 tháng	1 năm	1 thế kỷ
$\lg n$							
$\sqrt{n}$							
n							
$n \lg n$							
n^2							
n^3							
2^n							
$n!$							

1-2 Sắp xếp chèn trên các mảng nhỏ trong sắp xếp trộn

Mặc dù phương pháp sắp xếp trộn chạy trong $\Theta(n \lg n)$ thời gian trường hợp xấu nhất và sắp xếp chèn chạy trong $\Theta(n^2)$ thời gian trường hợp xấu nhất, các thừa số bất biến trong sắp xếp chèn khiến nó chạy nhanh hơn với n nhỏ. Do đó, tốt nhất ta nên sử dụng kỹ thuật sắp xếp chèn bên trong sắp xếp trộn khi các bài toán con trở nên đủ nhỏ. Xem xét một dạng cải biên kỹ thuật sắp xếp trộn ở đó n/k danh sách con có chiều dài k được sắp xếp bằng kỹ thuật sắp xếp chèn rồi được hợp nhất bằng cơ chế trộn chuẩn, ở đó k là một giá trị sẽ được xác định.

a. Chứng tỏ n/k danh sách con, mỗi danh sách có chiều dài k, có thể được sắp xếp bằng phương pháp sắp xếp chèn trong $\Theta(nk)$ thời gian trường hợp-xấu nhất.

b. Chứng tỏ các danh sách con có thể được trộn trong $\Theta(n \lg(n/k))$ thời gian ca xấu nhất.

c. Cho thuật toán đã sửa đổi chạy trong $\Theta(nk + n \lg(n/k))$ thời gian ca xấu nhất, nêu giá trị tiệm cận lớn nhất của k (hệ ký hiệu Θ) dưới dạng một hàm n mà thuật toán đã sửa đổi có cùng thời gian thực hiện tiệm cận dưới dạng sắp xếp trộn chuẩn?

d. k được chọn ra sao trong thực tế?

1-3 Các phép nghịch đảo

Cho $A[1..n]$ là một mảng n con số riêng biệt. Nếu $i < j$ và $A[i] > A[j]$, thì cặp (i, j) được gọi là một ***phép nghịch đảo*** [inversion] của A.

a. Liệt kê năm phép nghịch đảo của mảng <2, 3, 8, 6, 1>.

b. Mảng nào với các thành phần của tập hợp $\{1, 2, \ldots, n\}$ có nhiều phép nghịch đảo nhất? Có bao nhiêu?

c. Nêu mối quan hệ giữa thời gian thực hiện của kỹ thuật sắp xếp chèn và số lượng phép nghịch đảo trong mảng đầu vào? Xác minh câu trả lời.

d. Cho một thuật toán xác định số lượng phép nghịch đảo trong một phép hoán vị bất kỳ trên n thành phần trong $\Theta(n \lg n)$ thời gian ca xấu nhất. (*Mách nước:* Sửa đổi phương pháp sắp xếp trộn.)

Ghi chú Chương

Có nhiều tài liệu tuyệt vời nói về chủ đề thuật toán nói chung, kể cả các tài liệu của Aho, Hopcroft, và Ullman [4, 5], Baase [14], Brassard và Bratley [33], Horowitz và Sahni [105], Knuth [121, 122, 123], Manber [142], Mehlhorn [144, 145, 146], Purdom và Brown [164], Reingold, Nievergelt, và Deo [167], Sedgewick [175], và Wilf [201]. Bentley [24, 25] và Gonnet [90] có mô tả vài khía cạnh thực tiễn hơn về thiết kế thuật toán.

Vào năm 1968, Knuth xuất bản tập đầu tiên trong ba tập có tiêu đề chung là *The Art of Computer Programming* [121, 122, 123]. Tập đầu tiên đã đánh dấu sự khởi đầu trong tiến trình nghiên cứu hiện đại các thuật toán máy tính tập trung vào việc phân tích thời gian thực hiện, và

trọn bộ là tài liệu tham khảo quý giá và hấp dẫn đối với nhiều chủ đề trình bày ở đây. Theo Knuth, từ "algorithm" xuất phát từ tên "al-Khowarizmi," một nhà toán học Ba-tư vào thế kỷ thứ chín.

Aho, Hopcroft, và Ullman [4] đã ủng hộ phương pháp phân tích tiệm cận của các thuật toán như một biện pháp để so sánh khả năng thực hiện tương đối. Họ cũng đã phổ biến cách dùng các quan hệ truy toán để mô tả các thời gian thực hiện của các thuật toán đệ quy.

Knuth [123] cung cấp một cách giải quyết bách khoa của nhiều thuật toán sắp xếp. Phép so sánh các thuật toán sắp xếp của ông ta (trang 421) có gộp các tiến trình phân tích đếm bước chính xác, giống như kiểu mà ta đã thực hiện ở đây đối với phương pháp sắp xếp chèn. Phần mô tả của Knuth về sắp xếp chèn bao hàm vài biến thể của thuật toán. Mà quan trọng nhất là kiểu sắp xếp Shell [Shell's sort], do D. *L.* Shell giới thiệu, sử dụng kỹ thuật sắp xếp chèn trên các dãy con định kỳ của đầu vào để tạo ra một thuật toán sắp xếp nhanh hơn.

Sắp xếp trộn cũng được Knuth mô tả. Ông cho biết một máy so phiếu đục lỗ cơ học có khả năng trộn hai cỗ bài gồm các lá bài đục lỗ trong một lượt xếp đã được sáng chế vào năm 1938. J. von Neumann, một trong những người tiên phong về khoa học máy tính, dường như đã viết một chương trình cho kỹ thuật sắp xếp trộn trên máy tính EDVAC vào năm 1945.

I Căn bản về Toán học

Giới thiệu

Tiến trình phân tích các thuật toán thường yêu cầu ta cậy vào nhiều công cụ toán học. Một số công cụ chỉ đơn giản là đại số cấp trung học, song một số khác, như giải quyết các phép truy toán, có thể còn mới đối với bạn. Phần này của cuốn sách tóm lược các phương pháp và các công cụ mà ta sẽ dùng để phân tích các thuật toán. Nó được tổ chức chủ yếu để tham khảo, tuy có vài chủ đề được gia cố thiên về giáo trình.

Đừng nên ngốn hết phần toán học này ngay một lúc. Hãy lướt qua các chương trong phần này để xem chúng chứa nội dung gì. Sau đó, bạn có thể tiến thẳng đến các chương tập trung vào các thuật toán. Và khi đọc các chương đó, bạn sẽ tham khảo lại phần này mỗi khi cần hiểu rõ hơn về các công cụ được dùng trong các tiến trình phân tích toán học. Tuy nhiên, đến một lúc nào đó, bạn sẽ thấy cần nghiên cứu trọn từng chương này, để có một nền móng vững chắc về các kỹ thuật toán học.

Chương 2 định nghĩa chính xác vài hệ ký hiệu tiệm cận, một ví dụ đó là hệ ký hiệu Θ mà bạn đã gặp trong Chương 1. Phần còn lại của Chương 2 chủ yếu trình bày hệ ký hiệu toán học. Mục tiêu của nó là để bảo đảm bạn dùng hệ ký hiệu khớp với cuốn sách này chứ không chủ trương giới thiệu các khái niệm toán học mới.

Chương 3 đưa ra các phương pháp để đánh giá và định cận các phép lấy tổng, thường xảy ra trong khi phân tích các thuật toán. Nhiều công thức trong chương này đều có trong các sách giáo khoa tính toán, song bạn sẽ thấy tiện dụng khi các phương pháp này được biên soạn chung một chỗ ở đây.

Chương 4 mô tả các phương pháp giải quyết các phép truy toán, mà ta đã dùng để phân tích kỹ thuật sắp xếp trộn trong Chương 1 và sẽ gặp lại nhiều lần. Một kỹ thuật mạnh đó là "phương pháp chủ ["master method"], có thể dùng để giải quyết các phép truy toán nảy sinh từ các thuật toán chia để trị. Phần lớn Chương 4 dành để chứng minh tính đúng đắn của phương pháp chủ, do đó bạn có thể bỏ qua mà không hại gì.

Chương 5 chứa các định nghĩa căn bản và các ký hiệu về các tập hợp, các quan hệ, các hàm, các đồ thị, và các cây. Chương này cũng đưa ra vài tính chất căn bản của các đối tượng toán học này. Đây là phần nội dung thiết yếu để hiểu rõ tập giáo trình này, song có thể an tâm bỏ qua nếu như bạn đã theo một khóa học về toán rời rạc.

Chương 6 bắt đầu bằng các nguyên lý cơ bản về đếm: các phép hoán vị, các tổ hợp, vân vân. Phần còn lại của chương mô tả các định nghĩa và các tính chất của xác suất căn bản. Hầu hết các thuật toán trong cuốn sách này không yêu cầu xác suất để phân tích, và do đó bạn có thể bỏ qua các đoạn sau của chương trong lần đọc đầu tiên, thậm chí chẳng cần lướt qua chúng. Về sau, khi gặp một phân tích xác suất cần tìm hiểu kỹ hơn, bạn quay lại Chương 6 để tham khảo.

2 Sự Tăng trưởng của Các hàm

Cấp tăng trưởng thời gian thực hiện của một thuật toán, được định nghĩa trong Chương 1, mô tả sơ bộ đặc trưng hiệu năng của thuật toán và cũng cho phép ta so sánh khả năng thực hiện tương đối của các thuật toán khác. Một khi kích cỡ đầu vào n trở nên đủ lớn, phương pháp sắp xếp trộn, với $\Theta(n \lg n)$ thời gian thực hiện trường hợp xấu nhất của nó, sẽ đánh bạt phương pháp sắp xếp chèn, có thời gian thực hiện trường hợp xấu nhất là $\Theta(n^2)$. Mặc dù đôi lúc có thể xác định thời gian thực hiện chính xác của một thuật toán, như trường hợp sắp xếp chèn trong Chương 1, song độ chính xác phụ trội thường không xứng với nỗ lực tính toán nó. Với các đầu vào đủ lớn, các hằng nhân [multiplicative constants] và số hạng cấp thấp của một thời gian thực hiện chính xác thường bị các hiệu ứng của chính kích cỡ đầu vào chi phối.

Khi xem xét các kích cỡ đầu vào đủ lớn để khiến chỉ mình cấp tăng trưởng thời gian thực hiện là thích đáng, ta đang nghiên cứu hiệu năng ***tiệm cận*** của các thuật toán. Nghĩa là, ta quan tâm đến cách gia tăng của thời gian thực hiện của một thuật toán có kích cỡ đầu vào *trong giới hạn,* bởi kích cỡ đầu vào gia tăng mà không có ranh giới. Thông thường, một thuật toán hiệu quả hơn về mặt tiệm cận sẽ là chọn lựa tốt nhất cho tất cả ngoại trừ các đầu vào rất nhỏ.

Chương này đưa ra vài phương pháp chuẩn để đơn giản hóa cách phân tích tiệm cận của các thuật toán. Đoạn kế tiếp bắt đầu bằng cách định nghĩa vài kiểu "hệ ký hiệu tiệm cận," mà ta đã thấy một ví dụ trong hệ ký hiệu Θ. Sau đó trình bày vài quy ước ký hiệu dùng trong suốt cuốn sách này, và cuối cùng ta ôn lại cách ứng xử của các hàm thường nảy sinh trong khi phân tích các thuật toán.

2.1 Hệ ký hiệu tiệm cận

Các ký hiệu mà ta dùng để mô tả thời gian thực hiện tiệm cận của một thuật toán được định nghĩa theo dạng các hàm có các lĩnh vực là tập hợp các số tự nhiên **N** = {0, 1, 2,...}. Các ký hiệu như vậy tỏ ra tiện dụng khi mô tả hàm thời gian thực hiện ca xấu nhất $T(n)$, thường chỉ được

định nghĩa dựa trên các kích cỡ đầu vào số nguyên. Tuy nhiên, đôi lúc cũng tỏ ra tiện dụng khi *lạm dụng* hệ ký hiệu tiệm cận theo nhiều cách khác nhau. Ví dụ, có thể dễ dàng mở rộng hệ ký hiệu theo lĩnh vực các số thực hoặc, một cách khác, hạn chế theo một tập hợp con các số tự nhiên. Tuy nhiên, điều quan trọng là hiểu rõ ý nghĩa đích thực của hệ ký hiệu sao cho khi bị lạm dụng, nó không bị *dùng sai*. Đoạn này định nghĩa các ký hiệu tiệm cận căn bản và giới thiệu vài cách lạm dụng phổ biến.

Hệ ký hiệu Θ

Chương 1 đã giới thiệu thời gian thực hiện trường hợp xấu nhất của phương pháp sắp xếp chèn là $T(n) = \Theta(n^2)$. Giờ đây ta định nghĩa ý nghĩa của hệ ký hiệu này. Với một hàm đã cho $g(n)$, qua $\Theta(g(n))$ ta thể hiện *tập hợp các hàm*

$\Theta(g(n)) = \{f(n)$: ở đó tồn tại các hằng dương c_1, c_2, và n_0 sao cho $0 \leq c \lg(n) \leq f(n) \leq c_2 g(n)$ với tất cả $n \geq n_0\}$.

Một hàm $f(n)$ thuộc về tập hợp $\Theta(g(n))$ nếu ở đó tồn tại các hằng dương c_1, và c_2 sao cho nó có thể "được kẹp" giữa $c_1 g(n)$ và $c_2 g(n)$, với n đủ lớn. Mặc dù $\Theta(g(n))$ là một tập hợp, ta viết "$f(n) = \Theta(g(n))$" để nêu rõ $f(n)$ là một phần tử của $\Theta(g(n))$, hoặc "$f(n) \in \Theta(g(n))$." Thoạt tiên, kiểu lạm dụng đẳng thức để thể hiện sự thuộc về tập hợp này có vẻ như dễ gây lẫn lộn, song nó có các ưu điểm mà ta sẽ thấy ở phần sau trong đoạn này.

Hình 2.1 (a) minh họa trực quan các hàm $f(n)$ và $g(n)$, ở đó $f(n) = \Theta(g(n))$. Với tất cả giá trị n về bên phải của n_0, giá trị $f(n)$ nằm tại hoặc bên trên $c_1 g(n)$ và tại hoặc bên dưới $c_2 g(n)$. Nói cách khác, với tất cả $n \geq n_0$, hàm $f(n)$ bằng $g(n)$ bên trong một thừa số bất biến. Ta nói $g(n)$ là một ***cận sát theo tiệm cận*** [asymptotically tight bound] của $f(n)$.

Phần định nghĩa $\Theta(g(n))$ yêu cầu mọi phần tử của $\Theta(g(n))$ là ***không âm theo tiệm cận*** [asymptotically nonnegative], nghĩa là, $f(n)$ đó không âm mỗi khi n đủ lớn. Kết quả là, chính hàm $g(n)$ phải là không âm theo tiệm cận, bằng không tập hợp $\Theta(g(n))$ là trống. Do đó ta mặc nhận rằng mọi hàm được dùng trong hệ ký hiệu Θ sẽ không âm theo tiệm cận. Giả thiết này cũng áp dụng cho các ký hiệu tiệm cận khác được định nghĩa trong chương này.

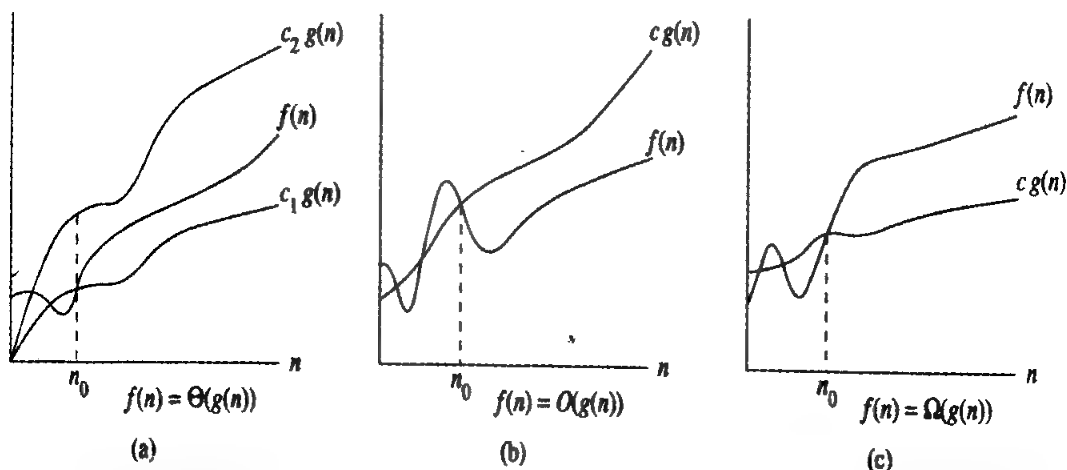
Chương 1 đã giới thiệu một khái niệm không chính thức về hệ ký hiệu 0 mà chung quy là vứt bỏ các số hạng cấp thấp và bỏ qua hệ số đầu của số hạng cấp cao nhất. Hãy xác minh ngắn gọn trực giác này bằng cách dùng phần định nghĩa hình thức để chứng tỏ $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. Để thực hiện, phải xác định các hằng dương c_1, c_2, và n_0 sao cho

$c_1n^2 \le \frac{1}{2}n^2 - 3n \le c_2n^2$

với tất cả $n \ge n_0$. Chia với n^2 cho ra

$c_1 \le \frac{1}{2} - \frac{3}{n} \le c_2$

Có thể tạo bất đẳng thức tay phải [right-hand inequality] để áp dụng cho bất kỳ giá trị $n \ge 1$ bằng cách chọn $c_2 \ge 1/2$. Cũng vậy, có thể tạo bất đẳng thức tay trái để cho bất kỳ giá trị $n \ge 7$ bằng cách chọn $c_1 \le 1/14$. Như vậy, nhờ chọn $c1 = 1/14$, $c_2 = 1/2$, và $n^0 = 7$, ta có thể xác minh $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. Tất nhiên, vẫn có các chọn lựa khác cho các hằng, song điều quan trọng đó là việc tồn tại một khả năng chọn lựa nào đó. Lưu ý, các hằng này tùy thuộc vào hàm $\frac{1}{2}n^2 - 3n$; một hàm khác thuộc về $\Theta(n^2)$ thường yêu cầu các hằng khác nhau.



Hình 2.1 Các ví dụ đồ họa của các ký hiệu Θ, O, và Ω . Trong mỗi phần, giá trị n_0 đã nêu là giá trị tối thiểu khả dĩ; các giá trị lớn hơn cũng làm việc. (a) hệ ký hiệu Θ định cận một hàm nằm trong các thừa số bất biến. Ta viết $f(n) = \Theta(g(n))$ nếu ở đó tồn tại các hằng dương n_0, c_1, và c_2 sao cho về bên phải của n_0, giá trị $f(n)$ luôn nằm giữa $c_1g(n)$ và $c_2g(n)$ kể luôn cả hai giá trị này. (b) hệ ký hiệu O cung cấp một cận trên cho một hàm nằm trong một thừa số bất biến. Ta viết $f(n) = O(g(n))$ nếu có các hằng dương n_0 và c sao cho về bên phải của no, giá trị $f(n)$ luôn nằm tại hoặc bên dưới $cg(n)$. (c) hệ ký hiệu Ω đưa ra một cận dưới cho một hàm nằm trong một thừa số bất biến. Ta viết $f(n) = \Omega(g(n))$ nếu có các hằng dương n_o và c sao cho về bên phải của n_0, giá trị $f(n)$ luôn nằm tại hoặc bên trên $cg(n)$.

Cũng có thể dùng phần định nghĩa hình thức để xác minh $6n^3 \ne \Theta(n^2)$. Giả sử, với mục đích trái ngược, c_2 và n_0 tồn tại sao cho $6n^3 \le C_2n^2$ với tất cả $n \ge n_0$. Nhưng rồi $n \le c_2/6$, không thể áp dụng cho n lớn một cách tùy ý, bởi c_2 là bất biến.

Theo trực giác, các số hạng cấp thấp của một hàm dương tiệm cận có thể được bỏ qua trong khi xác định các cận sát tiệm cận bởi chúng không quan trọng đối với n lớn. Một phân số bé của số hạng cấp cao nhất là đủ để chi phối các số hạng cấp thấp. Do đó, việc ấn định c_1 theo một giá trị hơi nhỏ hơn hệ số của số hạng cấp cao nhất và việc ấn định c_2 theo một giá trị hơi lớn hơn sẽ cho phép thỏa các bất đẳng thức trong định nghĩa của hệ ký hiệu Θ. Cũng vậy, ta có thể bỏ qua hệ số của số hạng cấp cao nhất, bởi nó chỉ thay đổi c_1 và c_2 theo một thừa số bất biến bằng với hệ số đó.

Ví dụ, xét một hàm bậc hai $f(n) = an^2 + bn + c$, ở đó a, b, và c là các hằng và $a > 0$. Vứt bỏ các số hạng cấp thấp và bỏ qua hằng sẽ cho ra $f(n) = \Theta(n^2)$. Về hình thức, để nêu cùng nội dung, ta lấy các hằng $c_1 = a/4$, $c_2 = 7a/4$, và $n_0 = 2 . \max((|b|/a), \sqrt{(|c|/a)})$. Người đọc có thể xác minh rằng $0 \le c_1 n^2 \le an^2 + bn + c \le c_2 n^2$ với tất cả $n \ge n_0$. Nói chung, với bất kỳ đa thức $p(n) = \sum_{i=0}^{d} a_i n^i$, ở đó a_i là các hằng và $a_d > 0$, ta có $p(n) = \Theta(n^d)$ (xem Bài toán 2-1).

Bởi một hằng bất kỳ là một đa thức bậc 0, nên ta có thể diễn tả một hàm bất biến dưới dạng $\Theta)(n^0)$, hoặc $\Theta(1)$. Tuy nhiên, hệ ký hiệu sau là một lạm dụng nhỏ, bởi ta không rõ biến nào đang hướng tới vô cực 1. Ta thường dùng hệ ký hiệu $\Theta(1)$ để chỉ một hằng hoặc một hàm bất biến đối với một biến nào đó.

Hệ ký hiệu *O*

Xét về tiệm cận, hệ ký hiệu Θ định cận một hàm từ bên trên và bên dưới. Khi chỉ có một ***cận trên tiệm cận*** [asymptotic higher bound], ta dùng hệ ký hiệu 0. Với một hàm đã cho $g(n)$, bằng $O(g(n))$ ta thể hiện tập hợp các hàm

$O(g(n)) = \{f(n) :$ ở đó tồn tại các hằng dương c và n_0 sao cho

$0 \le f(n) \le cg(n)$ với tất cả $n \ge n_0\}$.

Ta dùng hệ ký hiệu 0 để cung cấp một cận trên trên một hàm, trong phạm vi một thừa số bất biến. Hình 2.1(**b**) nêu trực giác đàng sau hệ ký hiệu 0. Với tất cả giá trị n về bên phải của n_0, giá trị của hàm $f(n)$ là tại hoặc bên dưới $g(n)$.

---

1 Vấn đề thực sự đó là hệ ký hiệu bình thường của ta dành cho các hàm lại không phân biệt các hàm với các giá trị. Trong hệ tính toán λ[A-calculus], các tham số cho một hàm được chỉ định rõ ràng: hàm n^2 có thể được viết là $\lambda n.n^2$, hoặc thậm chí $\lambda r.r^2$. Tuy nhiên, việc thừa nhận một hệ ký hiệu nghiêm ngặt hơn sẽ làm phức tạp các kiểu vận dụng đại số, và do đó ta quyết định dung nạp kiểu lạm dụng.

Để nêu rõ một hàm $f(n)$ là một phần tử của $O(g(n))$, ta viết $f(n) = O(g(n))$. Lưu ý, $f(n) = \Theta(g(n))$ hàm ý $f(n) = O(g(n))$, bởi hệ ký hiệu Θ mạnh hơn hệ ký hiệu O. Trên lý thuyết thành văn, ta có $\Theta(g(n)) \subseteq O(g(n))$. Do đó, chứng minh của chúng ta rằng mọi hàm bậc hai $an^2 + bn + c$, ở đó $a > 0$, nằm trong $\Theta(n^2)$ cũng cho thấy mọi hàm bậc hai nằm trong $O(n^2)$. Điều ngạc nhiên hơn đó là mọi hàm *tuyến tính* $an + b$ nằm trong $O(n^2)$, được xác minh dễ dàng bằng cách lấy $c = a + |b|$ và $n_0 = 1$.

Một số bạn đọc đã từng gặp hệ ký hiệu O trước đây có thể thấy hơi lạ khi chúng tôi viết, chẳng hạn, $n = O(n^2)$. Trong tài liệu giáo khoa, hệ ký hiệu O đôi lúc được dùng không chính thức để mô tả các cận sát tiệm cận, nghĩa là, nội dung mà ta đã định nghĩa bằng hệ ký hiệu Θ. Tuy nhiên, trong cuốn sách này, khi viết $f(n) = O(g(n))$, ta đơn thuần xác nhận một bội số bất biến nào đó của $g(n)$ là một cận trên tiệm cận trên $f(n)$, mà không xác nhận một cận trên sát đến mức nào. Việc phân biệt các cận trên tiệm cận với các cận sát tiệm cận giờ đây đã trở thành chuẩn trong các tài liệu thuật toán.

Dùng hệ ký hiệu O, ta thường có thể mô tả thời gian thực hiện của một thuật toán bằng cách đơn thuần xem xét kỹ cấu trúc chung của thuật toán. Ví dụ, cấu trúc vòng lặp lồng đôi của thuật toán sắp xếp chèn trong Chương 1 lập tức cho ra một cận trên $O(n^2)$ trên thời gian thực hiện trường hợp xấu nhất: hao phí của vòng lặp phía trong được định cận từ bên trên theo $O(1)$ (hằng), cả hai chỉ số i và j đều tối đa là n, và vòng lặp phía trong được thi hành tối đa một lần cho mỗi trong số n^2 cặp giá trị của i và j.

Bởi hệ ký hiệu O mô tả một cận trên, nên khi ta dùng nó để định cận thời gian thực hiện trường hợp xấu nhất của một thuật toán, ta cũng ngụ ý định cận thời gian thực hiện của thuật toán trên các đầu vào tùy ý. Như vậy, cận $O(n^2)$ trên thời gian thực hiện trường hợp xấu nhất của phương pháp sắp xếp chèn cũng áp dụng cho thời gian thực hiện của nó trên mọi đầu vào. Tuy nhiên, cận $\Theta(n^2)$ trên thời gian thực hiện trường hợp xấu nhất của thuật toán sắp xếp chèn không hàm ý một cận $\Theta(n^2)$ trên thời gian thực hiện của tiến trình sắp xếp chèn trên *mọi* đầu vào. Ví dụ, trong Chương 1 ta đã thấy rằng khi đầu vào đã được xếp sẵn, tiến trình sắp xếp chèn chạy trong thời gian $\Theta(n)$.

Về mặt kỹ thuật, là một lạm dụng khi nói thời gian thực hiện của sắp xếp chèn là $O(n^2)$, bởi với một n đã cho, thời gian thực hiện thực tế tùy thuộc vào đầu vào cụ thể có kích cỡ n. Nghĩa là, thời gian thực hiện không thực sự là một hàm của n. Điều mà ta ngụ ý khi nói "thời gian thực hiện là $O(n^2)$" đó là thời gian thực hiện trường hợp xấu nhất (là một hàm của n) là $O(n^2)$, hoặc tương đương, bất kể đã chọn một đầu vào

cụ thể có kích cỡ n nào cho mỗi giá trị của n, thời gian thực hiện trên tập hợp các đầu vào đó vẫn là $O(n^2)$.

Hệ ký hiệu Ω

Giống như hệ ký hiệu O cung cấp một cận trên tiệm cận trên một hàm, hệ ký hiệu Ω cung cấp một ***cận dưới tiệm cận*** [asymptotic lower bound]. Với một hàm đã cho $g(n)$, bằng $\Omega(g(n))$ ta thể hiện tập hợp các hàm

$$\Omega(g(n)) = \{f(n) : \text{ở đó tồn tại các hằng dương } c \text{ và } n_0 \text{ sao cho } 0 \leq cg(n) \leq f(n) \text{ với tất cả } n \geq n_0\}.$$

Hình 2. 1 (c) có nêu trực giác đàng sau hệ ký hiệu Ω. Với tất cả giá trị n về bên phải của n_0, giá trị của $f(n)$ nằm tại hoặc bên trên $g(n)$.

Từ các định nghĩa về các hệ ký hiệu tiệm cận mà ta đã gặp cho đến giờ, ta có thể dễ dàng chứng minh định lý quan trọng dưới đây (xem Bài tập 2.1-5).

Định lý 2.1

Với hai hàm bất kỳ $f(n)$ và $g(n)$, $f(n) = \Theta(g(n))$ nếu và chỉ nếu $f(n) = O(g(n))$ và $f(n) = \Omega(g(n))$.

Để lấy ví dụ về cách ứng dụng định lý này, chứng minh của chúng ta rằng $an^2 + bn + c = \Theta(n^2)$ với bất kỳ hằng a, b, và c, ở đó $a > 0$, cũng lập tức hàm ý rằng $an^2 + bn + c = \Omega(n^2)$ và $an^2 + bn + c = O(n^2)$. Trong thực tế, thay vì dùng Định lý 2.1 để có các cận trên và dưới tiệm cận từ các cận sát tiệm cận, như đã làm trong ví dụ này, ta thường dùng nó để chứng minh các cận sát tiệm cận từ các cận trên và dưới tiệm cận.

Do hệ ký hiệu Ω mô tả một cận dưới, khi ta dùng nó để định cận thời gian thực hiện trường hợp tốt nhất của một thuật toán, ta cũng ngụ ý định cận thời gian thực hiện của thuật toán trên các đầu vào tùy ý. Ví dụ, thời gian thực hiện trường hợp tốt nhất của sắp xếp chèn là $\Omega(n)$, hàm ý rằng thời gian thực hiện của sắp xếp chèn là $\Omega(n)$.

Như vậy, thời gian thực hiện của sắp xếp chèn rơi giữa $\Omega(n)$ và $O(n^2)$, bởi nó rơi tại bất kỳ đâu giữa một hàm tuyến tính của n và một hàm bậc hai của n. Hơn nữa, theo tiệm cận, các cận này càng sát càng tốt: ví dụ, thời gian thực hiện của sắp xếp chèn không phải là $\Omega(n^2)$, bởi tiến trình sắp xếp chèn chạy trong thời gian $\Theta(n)$ khi đầu vào được sắp xếp sẵn. Tuy nhiên, không phải là mâu thuẫn khi ta nói rằng thời gian thực hiện ca xấu nhất của sắp xếp chèn là $\Omega(n^2)$, bởi ở đó tồn tại một đầu vào khiến thuật toán chấp nhận thời gian $\Omega(n^2)$. Khi nói *thời gian thực hiện* (không có bổ từ) của một thuật toán là $\Omega(g(n))$, ta ngụ ý *bất kể đã chọn đầu vào cụ thể có kích cỡ n nào cho mỗi giá trị của n*, thời

gian thực hiện trên tập hợp các đầu vào đó ít nhất cũng là một hằng nhân $g(n)$, với n đủ lớn.

Hệ ký hiệu tiệm cận trong các phương trình

Ta đã thấy cách dùng hệ ký hiệu tiệm cận trong các công thức toán học. Ví dụ, trong khi giới thiệu hệ ký hiệu O, ta đã viết "$n = O(n^2)$." Ta cũng có thể viết $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$. Làm sao để diễn dịch các công thức như vậy?

Khi hệ ký hiệu tiệm cận đứng một mình bên phía phải của một phương trình, như trong $n = O(n^2)$, ta đã định nghĩa sẵn dấu bằng để ám chỉ thuộc về tập hợp: $n \in O(n^2)$. Tuy nhiên, nói chung, khi hệ ký hiệu tiệm cận xuất hiện trong một công thức, ta diễn dịch nó dưới dạng thay cho một hàm nặc danh [anonymous function] nào đó mà ta không quan tâm đến tên. Ví dụ, công thức $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ có nghĩa là $2n^2 + 3n + 1 = 2n^2 + f(n)$, ở đó $f(n)$ là một hàm nào đó trong tập hợp $\Theta(n)$. Trong trường hợp này, $f(n) = 3n + 1$, mà quả thật nằm trong $\Theta(n)$.

Dùng hệ ký hiệu tiệm cận theo cách này có thể giúp loại bỏ chi tiết không cần thiết và khả năng gây bừa bộn trong một phương trình. Ví dụ, trong Chương 1 ta đã diễn tả thời gian thực hiện ca xấu nhất của sắp xếp trộn dưới dạng phép truy toán

$$T(n) = 2T(n/2) + \Theta(n) .$$

Nếu ta chỉ quan tâm đến cách ứng xử tiệm cận của $T(n)$, chẳng có ý nghĩa gì khi đặc tả chính xác tất cả số hạng cấp thấp; chúng được hiểu ngầm là nằm trong hàm nặc danh do số hạng $\Theta(n)$ thể hiện.

Số lượng hàm nặc danh trong một biểu thức được hiểu ngầm là bằng số lần xuất hiện của hệ ký hiệu tiệm cận. Ví dụ, trong biểu thức

$$\sum_{i=1}^{n} o(i),$$

chỉ có độc một hàm nặc danh (một hàm của i). Do đó, kiểu diễn tả này *không giống* như $O(1) + O(2) + O(n)$, không thực sự có một kiểu diễn dịch không lỗi.

Trong vài trường hợp, hệ ký hiệu tiệm cận xuất hiện bên phía trái của một phương trình, như trong

$$2n^2 + \Theta(n) = \Theta(n^2)$$

Ta diễn dịch các phương trình như vậy bằng quy tắc sau: *Bất kể cách chọn các hàm nặc danh bên phía trái của dấu bằng, ta cũng có một cách để chọn các hàm nặc danh bên phía phải của dấu bằng để khiến phương trình hợp lệ*. Do đó, ý nghĩa của ví dụ đó là với *bất kỳ* hàm $f(n)$

$\in \Theta(n)$, ta cũng có một hàm $g(n) \in \Theta(n^2)$ *nào đó* sao cho $2n^2 + f(n) = g(n)$ với tất cả mọi n. Nói cách khác, bên phía phải của một phương trình cho ta cấp chi tiết thô hơn so với bên phía trái.

Một số quan hệ như vậy có thể kết xích với nhau, như trong

$$2n^2 + 3n + 1 = 2n^2 + \Theta(n)$$
$$= \Theta(n^2).$$

Ta có thể diễn dịch mỗi phương trình riêng biệt bằng quy tắc trên đây. Phương trình đầu tiên nói rằng có một hàm $f(n) \in \Theta(n)$ *nào đó* sao cho $2n^2 + 3n + 1 = 2n^2 + f(n)$ với tất cả mọi n. Phương trình thứ hai nói rằng với *bất kỳ* hàm $g(n) \in \Theta(n)$ (như trường hợp $f(n)$ trên đây), ta có một hàm $h(n) \in \Theta(n^2)$ *nào đó* sao cho $2n^2 + g(n) = h(n)$ với tất cả mọi n. Lưu ý, kiểu diễn dịch này hàm ý rằng $2n^2 + 3n + 1 = \Theta(n^2)$, là nội dung mà tiến trình kết xích các phương trình cung cấp theo trực giác.

Hệ ký hiệu o

Cận trên tiệm cận mà hệ ký hiệu O có thể hoặc không thể sát theo tiệm cận [asymptotically tight]. Cận $2n^2 = O(n^2)$ là sát theo tiệm cận, nhưng cận $2n = O(n^2)$ thì không. Ta dùng hệ ký hiệu o để thể hiện một cận trên không sát theo tiệm cận. Về hình thức, ta định nghĩa $o(g(n))$ ("o nhỏ của g của n") dưới dạng tập hợp

$o(g(n)) = \{f(n)$: với mọi hằng dương $c > 0$, ở đó tồn tại một hằng $n_0 > 0$ sao cho $0 \leq f(n) < cg(n)$ với tất cả $n \geq n_0\}$.

Ví dụ, $2n = o(n^2)$, nhưng $2n^2 \neq o(n^2)$.

Các định nghĩa về hệ ký hiệu O và hệ ký hiệu o cũng tương tự. Khác biệt chính đó là trong $f(n) = O(g(n))$, cận $0 \leq f(n)$:5 $cg(n)$ áp dụng cho một hằng $c > 0$ *nào đó*, nhưng trong $f(n) = o(g(n))$, cận $0 \leq f(n) < cg(n)$ áp dụng cho *tất cả* mọi hằng $c > 0$. Về trực giác, trong hệ ký hiệu o, hàm $f(n)$ trở thành không quan trọng tương đối với $g(n)$ khi n tiến gần đến vô cực; nghĩa là,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \quad (2.1)$$

Có vài tác giả dùng giới hạn này như một định nghĩa của hệ ký hiệu o; phần định nghĩa trong cuốn sách này cũng hạn chế các hàm nặc danh là không âm theo tiệm cận.

Hệ ký hiệu ω

Để liên tưởng, hệ ký hiệu ω với hệ ký hiệu Ω giống như hệ ký hiệu o với hệ ký hiệu O. Ta dùng hệ ký hiệu ω để thể hiện một cận dưới

không sát theo tiệm cận. Một cách để định nghĩa nó đó là bằng

$f(n) \in \omega(g(n))$ nếu và chỉ nếu $g(n) \in o(f(n))$.

Tuy nhiên, về hình thức, ta định nghĩa $\omega(g(n))$ ("ômêga nhỏ của g của n") dưới dạng tập hợp

$\omega(g(n)) = \{f(n)$: với mọi hằng dương $c > 0$, ở đó tồn tại một hằng $n_0 > 0$ sao cho $0 \leq cg(n) < f(n)$ với tất cả $n \geq n_0\}$.

Ví dụ, $n^2/2 = (\omega(n)$, nhưng $n^2/2 \neq \omega(n^2)$. Quan hệ $f(n) = \omega(g(n))$ hàm ý

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty,$$

nếu giới hạn tồn tại. Nghĩa là, $f(n)$ trở nên lớn tùy ý tương đối với $g(n)$ khi n tiến gần đến vô cực.

So sánh các hàm

Nhiều tính chất quan hệ của các số thực cũng áp dụng cho các phép so sánh tiệm cận. Dưới đây, mặc nhận rằng $f(n)$ và $g(n)$ là dương theo tiệm cận.

Tính bắc cầu:

$f(n) = \Theta(g(n))$ và $g(n) = \Theta(h(n))$ hàm ý $f(n) = \Theta(h(n))$,

$f(n) = O(g(n))$ và $g(n) = O(n(n))$ hàm ý $f(n) = O(h(n))$,

$f(n) = \Omega(g(n))$ và $g(n) = \Omega(n(n))$ hàm ý $f(n) = \Omega(h(n))$,

$f(n) = o(g(n))$ và $g(n) = o(n(n))$ hàm ý $f(n) = o(h(n))$,

$f(n) = \omega(g(n))$ và $g(n) = \omega(n(n))$ hàm ý $f(n) = \omega(h(n))$,

Tính phản chiếu:

$f(n) = \Theta(f(n))$,

$f(n) = O(f(n))$,

$f(n) = \Omega(f(n))$.

Tính đối xứng:

$f(n) = \Theta(g(n))$ nếu và chỉ nếu $g(n) = \Theta(f(n))$.

Chuyển vị tính đối xứng:

$f(n) = 0(g(n))$ nếu và chỉ nếu $g(n) = \Omega(f(n))$

$f(n) = o(g(n))$ nếu và chỉ nếu $g(n) = \omega(f(n))$

Do các tính chất này áp dụng cho các ký hiệu tiệm cận, nên ta có thể rút ra một tương tự giữa phép so sánh tiệm cận của hai hàm f và g với phép so sánh hai số thực a và b:

$f(n) = O(g(n)) \quad \approx \quad a \leq b$,

$f(n) = \Omega(g(n)) \quad \approx \quad a \geq b$,

$f(n) = \Theta(g(n)) \quad \approx \quad a = b$,

$f(n) = o(g(n)) \quad \approx \quad a < b$,

$f(n) = \omega(g(n)) \quad \approx \quad a > b$.

Tuy nhiên, một tính chất của các số thực không mang sang cho hệ ký hiệu tiệm cận:

Phép tam phân: Với hai số thực bất kỳ a và b, chính xác một trong các dạng sau đây phải được áp dụng: $a < b$, $a = b$, hoặc $a > b$.

Mặc dù có thể so sánh hai số thực bất kỳ, song không phải tất cả mọi hàm đều có thể so sánh được. Nghĩa là, với hai hàm $f(n)$ và $g(n)$, có thể có trường hợp không áp dụng $f(n) = O(g(n))$ hoặc $f(n) = \Omega(g(n))$. Ví dụ, các hàm n và $n^{1+\sin n}$ không thể so sánh bằng hệ ký hiệu tiệm cận, bởi giá trị của số mũ trong n $1+n^{1+\sin n}$ dao động giữa 0 và 2, tiếp nhận tất cả giá trị ở giữa.

Bài tập

2.1-1

Cho $f(n)$ và $g(n)$ là các hàm không âm theo tiệm cận. Dùng định nghĩa căn bản của hệ ký hiệu Θ, chứng minh rằng $\max(f(n), g(n)) = \Theta(f(n) + g(n))$.

2.1-2

Chứng tỏ với mọi hằng thực a và b, ở đó $b > 0$,

$$(n + a)^b = \Theta(n^b) \tag{2.2}$$

2.1-3

Giải thích tại sao lời phát biểu, "Thời gian thực hiện của thuật toán A ít nhất là $O(n^2)$", lại phi-nội dung [content-free].

2.1-4

$2^{n+1} = O(2^n)$? là $2^{2n} = O(2^n)$?

2.1-5

Chứng minh Định lý 2-1

2.1-6

Chứng minh rằng thời gian thực hiện của một thuật toán là $\Theta(g(n))$ nếu và chỉ nếu thời gian thực hiện trường hợp xấu nhất của nó là $O(g(n))$ và thời gian thực hiện trường hợp tốt nhất của nó là $\Omega(g(n))$.

2.1-7

Chứng minh $o(g(n)) \cap \omega)(g(n))$ là tập hợp trống.

2.1-8

Có thể mở rộng hệ ký hiệu của chúng ta theo trường hợp hai tham số n và m có thể tiến đến vô cực một cách độc lập theo các tốc độ khác nhau. Với một hàm đã cho $g(n, m)$, bằng $O(g(n, m))$ ta thể hiện tập hợp các hàm

$O(g(n, m)) = \{f(n, m)$: ở đó tồn tại các hằng dương c, n_0, và m_0

sao cho $0 \le f(n, m) \le cg(n, m)$

với tất cả $n \ge n_0$ và $m \ge m_0\}$.

Nêu các định nghĩa tương ứng của $\Omega(g(n, m))$ và $\Theta(g(n, m))$.

2.2 Các hệ ký hiệu chuẩn và các hàm chung

Đoạn này ôn lại vài hàm toán học và hệ ký hiệu chuẩn đồng thời khảo sát các mối quan hệ giữa chúng. Ngoài ra, ta cũng tìm hiểu cách dùng các hệ ký hiệu tiệm cận.

Tính đơn điệu

Một hàm $f(n)$ đang ***tăng đơn điệu*** nếu $m \le n$ hàm ý $f(m) \le f(n)$. Cũng vậy, nó đang ***giảm đơn điệu*** nếu $m \le n$ hàm ý $f(m) \ge f(n)$. Một hàm $f(n)$ đang ***tăng ngặt*** [strictly increasing] nếu $m < n$ hàm ý $f(m) < f(n)$ và đang ***giảm ngặt*** [strictly decreasing] nếu $m < n$ hàm ý $f(m) > f(n)$.

Cận dưới và cận trên

Với bất kỳ số thực x, ta dùng $\lfloor x \rfloor$ để thể hiện số nguyên lớn nhất nhỏ hơn hoặc bằng với x (gọi là "sàn của x") và $\lceil x \rceil$ để thể hiện số nguyên bé nhất lớn hơn hoặc bằng với x (gọi là "trần của x"). Với tất cả số thực x,

$$x - 1 < \lfloor x \rfloor \le x \le \lceil x \rceil < x + 1.$$

Với bất kỳ số nguyên n,

$$\lceil n/2 \rceil + \lfloor n/2 \rfloor = n,$$

và với bất kỳ số nguyên n và các số nguyên $a = 0$ và $b = 0$,

$$\lceil \lceil n/a \rceil /b \rceil = \lceil n/ab \rceil \tag{2.3}$$

và

$$\lfloor \lfloor n/a \rfloor /b \rfloor = \lfloor n/ab \rfloor \tag{2.4}$$

Các hàm sàn và trần tăng một cách đơn điệu.

Các đa thức

Căn cứ một số nguyên dương d, một ***đa thức trong n bậc d*** [polynomial in n of degree d] là một hàm $p(n)$ có dạng

$$p(n)=\sum_{i=0}^{n} a_i n^i.$$

ở đó các hằng $a_0, a_1, ..., a_d$ là *các **hệ số*** của đa thức và $a_d \neq 0$. Một đa thức là ***dương theo tiệm cận*** nếu và chỉ nếu $a_d > 0$. Với một đa thức dương theo tiệm cận $p(n)$ bậc d, ta có $p(n) = \Theta(n^d)$. Với bất kỳ hằng số thực $a \geq 0$, hàm n^a tăng một cách đơn điệu, và với bất kỳ hằng số thực $a \leq 0$, hàm n^a giảm một cách đơn điệu. Ta nói một hàm $f(n)$ được ***định cận theo đa thức*** nếu $f(n) = n^{O(1)}$, tương đương với phát biểu $f(n) = O(n^k)$ với một hằng k nào đó (xem Bài tập 2.2-2).

Các hàm số mũ

Với tất cả số thực $a \neq 0$, m, và n, ta có các đồng nhất thức sau đây:

$$\begin{aligned} a^0 &= 1, \\ a^1 &= a, \\ a^{-1} &= 1/a \\ (a^m)^n &= a^{mn} \\ (a^m)^n &= (a^n)^m \\ a^m a^n &= a^{m+n} \end{aligned}$$

Với tất cả n và $a \geq 1$, hàm a^n tăng một cách đơn điệu trong n.

Khi thuận tiện, ta sẽ mặc nhận $0^0 = 1$.

Các cấp tăng trưởng của các đa thức và các hàm số mũ có thể được liên kết bởi sự việc sau đây. Với tất cả các hằng số thực a và b sao cho $a > 1$.

$$\lim_{n\to\infty} \frac{n^b}{a^n} = 0,$$

từ đó có thể kết luận rằng

$$n^b = o(a^n)$$

Như vậy, các hàm số mũ dương tăng trưởng nhanh hơn các đa thức.

Dùng e để thể hiện 2.71828..., cơ số của hàm lôga tự nhiên, ta có với tất cả số thực x,

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{i=0}^{\infty} \frac{x^i}{i!},$$

ở đó "!" thể hiện hàm giai thừa, sẽ được định nghĩa sau trong đoạn này. Với tất cả số thực x, ta có bất đẳng thức

$$e^x \geq 1 + x, \tag{2.7}$$

ở đó đẳng thức chỉ áp dụng khi $x = 0$. Khi $|x| \leqslant 1$, ta có phép xấp xỉ

$$1 + x \leq e^x \leq 1 + x + x^2 \tag{2.8}$$

Khi $x \to 0$, phép xấp xỉ của e^x theo $1 + x$ là khá tốt:

$$e^x = 1 + x + \Theta)(x^2).$$

(Trong phương trình này, hệ ký hiệu tiệm cận được dùng để mô tả cách ứng xử giới hạn khi $x \to 0$ thay vì khi $x \to \infty$) Ta có với tất cả x,

$$\lim_{n \to \infty} \left(1 + \frac{x}{n}\right) = e^x.$$

Lôga

Ta sẽ dùng các hệ ký hiệu sau:

$\lg n = \log_2 n$ (lôga nhị phân),

$\ln n = \log_e n$ (lôga tự nhiên),

$\lg^k n = (\lg n)^k$ (mũ hóa),

$\lg\lg n = \lg(\lg n)$ (hợp thành).

Một quy ước ký hiệu quan trọng mà ta chấp nhận đó là *các hàm lôga sẽ chỉ áp dụng cho số hạng kế tiếp trong công thức*, sao cho $\lg n + k$ có nghĩa là $(\lg n) + k$ chứ không phải $\lg(n + k)$. Với $n > 0$ và $b > 1$, hàm $\log_b n$ tăng ngặt.

Với tất cả số thực $a > 0$, $b > 0$, $c > 0$, và n,

$$\begin{aligned}
a &= b^{\log a}, \\
\log_c(ab) &= \log_c a + \log_c b, \\
\log_b a^n &= n\log_b a, \\
\log_b a &= \frac{\log_c a}{\log_c b}, \\
\log_b(1/a) &= -\log_b a, \\
\log_b a &= \frac{1}{\log_a b}, \\
a^{\log_b n} &= n^{\log_b a}.
\end{aligned} \tag{2.9}$$

Do việc thay đổi cơ số của một lôga từ hằng này sang hằng khác chỉ làm thay đổi giá trị của lôga đó bằng một thừa số bất biến, nên ta thường

dùng hệ ký hiệu lg n" khi không quan tâm đến các thừa số bất biến, như trong hệ ký hiệu O. Các nhà khoa học máy tính thấy 2 là cơ số tự nhiên nhất cho lôga bởi có rất nhiều thuật toán và cấu trúc dữ liệu liên quan đến việc tách một bài toán thành hai phần.

Có một kiểu khai triển chuỗi đơn giản cho $\ln(1 + x)$ khi $|x| < 1$:

$$\ln(1 + x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \ldots$$

Ta cũng có các bất đẳng thức dưới đây cho $x > -1$:

$$\frac{x}{1 + x} \leq \ln(1 + x) \leq x, \qquad (2.10)$$

ở đó đẳng thức chỉ áp dụng cho $x = 0$.

Ta nói rằng một hàm $f(n)$ ***được định cận theo đa lôga*** [polylogarithmically bounded] nếu $f(n)$ $\lg^{O(1)} n$. Ta có thể liên kết sự tăng trưởng của các đa thức và đa lôga bằng cách dùng lg n thay cho n và 2^a cho a trong phương trình (2.5), cho ra

$$\lim_{n \to \infty} \frac{\lg^b n}{2^{a \lg n}} = \lim_{n \to \infty} \frac{\lg^b n}{n^a} = 0 .$$

Từ giới hạn này, ta có thể kết luận

$$\lg^b n = o(n^a)$$

với mọi hằng $a > 0$. Do đó, các hàm đa thức dương tăng trưởng nhanh hơn các hàm đa lôga.

Giai thừa

Hệ ký hiệu $n!$ (đọc là "n giai thừa" được định nghĩa cho các số nguyên $n \geq 0$ dưới dạng

$$n! = \begin{cases} 1 & \text{nếu } n = 0 , \\ n \cdot (n-1) & \text{nếu } n > 0 . \end{cases}$$

Do đó, $n! = 1.2.3 \ldots n$.

Một cận trên yếu trên hàm giai thừa là $n! \leq n^n$, bởi mỗi trong số n số hạng trong tích giai thừa tối đa là n. ***Phép xấp xỉ Stirling***,

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right),$$

ở đó e là cơ số của lôga tự nhiên, cho ta một cận trên sát hơn, và cả một cận dưới. Dùng phép xấp xỉ Stirling, ta có thể chứng minh

$$n! = o(n^n)$$

$n! = \omega(2n)$

$\lg(n!) = \Theta(n \lg n)$

Các cận dưới đây cũng áp dụng cho tất cả n:

$$\sqrt{2\pi n}\left(\frac{n}{e}\right)^n \le n! \le \sqrt{2\pi n}\left(\frac{n}{e}\right)^{n+1/12n} \qquad (2.12)$$

Hàm lôga lặp

Ta dùng hệ ký hiệu $\lg^* n$ (đọc là "log sao của n" [log star of n]) để thể hiện lôga lặp, được định nghĩa như sau. Cho hàm $\lg^{(i)} n$ được định nghĩa theo đệ quy cho các số nguyên không âm i dưới dạng

$$\lg^{(i)} n = \begin{cases} n & \text{nếu } i = 0, \\ \lg(\lg^{(i-1)} n) & \text{nếu } i > 0 \text{ và } \lg(i-1)\, n > 0, \\ \text{không xác định} & \text{nếu } i > 0 \text{ và } \lg(i-1)\, n \le 0 \text{ hoặc } \lg^{(i-1)} n \text{ là không xác định.} \end{cases}$$

Đừng quên phân biệt $\lg^{(i)} n$ (hàm lôga được ứng dụng i lần liên tiếp, bắt đầu bằng đối số n) từ $\lg^i n$ (lôga của n được nâng theo lũy thừa bậc i). Hàm lôga lặp được định nghĩa dưới dạng

$\lg^* n = \min\{ i \ge 0 : \lg^{(i)} n \le 1\}$.

Lôga lặp là một hàm tăng trưởng *rất* chậm:

$$\begin{aligned} \lg^* 2 &= 1, \\ \lg^* 4 &= 2, \\ \lg^* 16 &= 3, \\ \lg^* 65536 &= 4, \\ \lg^* (2^{65536}) &= 5. \end{aligned}$$

Bởi số lượng nguyên tử trong vũ trụ quan sát được được ước tính vào khoảng 10^{80}, nhỏ hơn nhiều so với 2^{65536}, nên hiếm khi ta gặp các giá trị của n sao cho $\lg^* n > 5$.

Các số Fibonacci

Các số Fibonacci được định nghĩa bởi phép truy toán sau:

$$\begin{aligned} F_0 &= 0, \\ F_i &= 1, \\ F_i &= F_{i-1} + F_{i-2} \quad \text{với } i \ge 2. \end{aligned} \qquad (2.13)$$

Do đó, mỗi số Fibonacci là tổng của hai số trước đó, cho ra dãy

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, … .

Các số Fibonacci có liên quan đến ***tỷ số vàng*** ϕ và liên hợp $\hat{\phi}$ của nó được các công thức dưới đây cung cấp:

$$\begin{aligned} \phi &= \frac{1+\sqrt{5}}{2} \\ &= 1.61803\ldots, \\ \hat{\phi} &= \frac{1-\sqrt{5}}{2} \\ &= -.61803\ldots \end{aligned} \qquad (2.14)$$

Cụ thể, ta có (2.15)

$$F_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}},$$

có thể được chứng minh bằng phương pháp quy nạp (Bài tập 2.2-7). Do $|\hat{\phi}| < 1$, nên ta có $|\hat{\phi}^i|/\sqrt{5} < 1/\sqrt{5} < 1/2$, sao cho số Fibonacci thứ i F_i bằng $\phi^i/\sqrt{5}$ được làm tròn theo số nguyên gần nhất. Do đó, các số Fibonacci tăng trưởng theo số mũ.

Bài tập

2.2-1

Chứng tỏ nếu $f(n)$ và $g(n)$ là các hàm tăng đơn điệu, thì các hàm $f(n) + g(n)$ và $f(g(n))$ cũng tăng như vậy, và ngoài ra nếu $f(n)$ và $g(n)$ là không âm, thì $f(n) \cdot g(n)$ tăng đơn điệu.

2.2-2

Dùng định nghĩa của hệ ký hiệu 0- để chứng tỏ $T(n) = n^{o(1)}$ nếu và chỉ nếu ở đó tồn tại một hằng $k > 0$ sao cho $T(n) = O(n^k)$.

2.2-3

Chứng minh phương trình (2.9).

2.2-4

Chứng minh $\lg(n!) = \Theta(n \lg n)$ và $n! = o(n^n)$.

2.2-5 *

Hàm $\lceil \lg n \rceil!$ có được định cận theo đa thức không? Hàm $\lceil \lg\lg n \rceil!$ có được định cận theo đa thức?

2.2-6 *

$\lg(\lg^* n)$ và $\lg^*(\lg n)$: cái nào lớn hơn theo tiệm cận?

2.2-7

Chứng minh bằng phương pháp quy nạp số Fibonacci thứ i thỏa đẳng thức $F_i = (\phi^i - \hat{\phi}^i)/\sqrt{5}$, ở đó ϕ là tỷ số vàng và $\hat{\phi}$ là liên hợp của nó.

2.2-8

Chứng minh rằng với $i \geq 0$, số Fibonacci $(i + 2)$ thỏa $F_{i+2} \geq \phi^i$.

Bài toán

2-1 Cách ứng xử tiệm cận của các đa thức

Cho

$$p(n) = \sum_{i=0}^{d} a_i n^i,$$

ở đó $a_d > 0$, là một đa thức bậc d trong n, và cho k là một hằng. Dùng định nghĩa của các hệ ký hiệu tiệm cận để chứng minh các tính chất dưới đây.

a. Nếu $k \geq d$, thì $p(n) = O(n^k)$.

b. Nếu $k \leq d$, thì $p(n) = \Omega(n^k)$.

c. Nếu $k = d$, thì $p(n) = \Theta(n^k)$.

d. Nếu $k > d$, thì $p(n) = o(n^k)$.

e. Nếu $k < d$, thì $p(n) = \omega(n^k)$.

2-2 Các mức tăng trưởng tiệm cận tương đối

Nêu rõ, với mỗi cặp biểu thức (A, B) trong bảng dưới đây, dấu A là O, o, Ω, ω, hoặc Θ của B. Giả sử $k \geq 1$, $\in > 0$, và $c > 1$ là các hằng. Đáp án của bạn sẽ theo dạng bảng có "yes" hoặc "no" viết trong mỗi hộp.

	A	B	O	o	Ω	w	Θ
a.	$\lg^k n$	n					
b.	n^k	c^n					
c.	$\sqrt{n}$	$n^{\sin n}$					
d.	2^n	$2^{n/2}$					
e.	$n^{\lg m}$	$m^{\lg n}$					
f.	$\lg(n!)$	$\lg(n^n)$					

2-3 Sắp xếp thứ tự bằng các cấp tăng trưởng tiệm cận

a. Phân loại các hàm dưới đây theo cấp tăng trưởng; nghĩa là, tìm một kiểu bố trí $g_1, g_2, \ldots, g_{30}$ của các hàm thỏa $g_1 = \Omega(g_2)$, $g_2 = \Omega(g_3)$, ..., $g_{29} = \Omega(g_{30})$. Phân hoạch danh sách thành các lớp tương đương sao cho $f(n)$ và $g(n)$ nằm trong cùng lớp nếu và chỉ nếu $f(n) = \Theta(g(n))$.

$\lg(\lg^* n)$	$2^{\lg^* n}$	$(\sqrt{2})\lg n$	n^2	$n!$	$(\lg n)!$
$(\frac{3}{2})^n$	n^3	$\lg^2 n$	$(\lg(n!)$	2^{2^n}	$n^{1/\lg n}$
$\ln\ln n$	$\lg^* n$	$n \,.\, 2n$	$n^{\lg\lg n}$	$\ln n$	1
$2^{\lg n}$	$(\lg n)^{\lg n}$	e^n	$4^{\lg n}$	$(n+1)!$	$\sqrt{\lg n}$

$\lg^*(\lg n)$ $\quad 2^{\sqrt{2\lg n}}$ $\quad n$ $\quad 2^n$ $\quad n\lg n$ $\quad 2^{2^{n+1}}$

b. Nêu một ví dụ về chỉ một hàm không âm $f(n)$ sao cho với tất cả các hàm $g_i(n)$ trong phần (a), $f(n)$ không phải $O(g_i(n))$ cũng không phải $\Omega(g_i(n))$.

2-4 Các tính chất hệ ký hiệu tiệm cận

Cho $f(n)$ và $g(n)$ là các hàm dương theo tiệm cận. Chứng minh hoặc bác bỏ từng giả định dưới đây.

a. $f(n) = O(g(n))$ hàm ý $g(n) = O(f(n))$.

b. $f(n) + g(n) = \Theta(\min(f(n), g(n)))$.

c. $f(n) = O(g(n))$ hàm ý $lg(f(n)) = O(lg(g(n)))$, ở đó $\lg(g(n)) > 0$ và $f(n) \geq 1$ với tất cả n đủ lớn.

d. $f(n) = O(g(n))$ hàm ý $2^{f(n)} = O(2^{g(n)})$

e. $f(n) = O(f(n))^2)$.

f. $f(n) = O(g(n))$ hàm ý $g(n) = \Omega(f(n))$.

g. $f(n) = \Theta(f(n/2))$.

h. $f(n) + o(f(n)) = \Theta(f(n))$.

2-5 Các biến thể của O và Ω

Có vài tác giả định nghĩa Ω theo cách hơi khác với ở đây; hãy dùng $\overset{\infty}{\Omega}$, (đọc là "vô hạn ômêga" cho kiểu định nghĩa thay thế này. Ta nói rằng $f(n) = \overset{\infty}{\Omega}(g(n))$ nếu ở đó tồn tại một hằng dương c sao cho $f(n) \geq cg(n) \geq 0$ với rất nhiều số nguyên n.

a. Chứng tỏ với hai hàm $f(n)$ và $g(n)$ bất kỳ không âm theo tiệm cận, $f(n) = O(g(n))$ hoặc $f(n) = \overset{\infty}{\Omega}(g(n))$ hoặc cả hai, trong khi đó điều này không đúng nếu ta dùng Ω thay vì Ω.

b. Mô tả các ưu và khuyết điểm tiềm ẩn khi dùng Ω thay vì Ω để đặc tả các thời gian thực hiện của các chương trình.

Có vài tác giả định nghĩa O theo cách hơi khác với ở đây; hãy dùng O' cho kiểu định nghĩa thay thế này. Ta nói rằng $f(n) = O'(g(n))$ nếu và chỉ nếu $|f(n)| = O(g(n))$.

c. Điều gì xảy ra cho từng hướng của "nếu và chỉ nếu' trong Định lý 2.1 theo định nghĩa mới này?

Có vài tác giả định nghĩa $\tilde{O}$ (đọc là "o mềm" [soft-oh]) để chỉ O có các thừa số lôga được bỏ qua:

$\tilde{O}(g(n)) = \{f(n)$: ở đó tồn tại các hằng dương c, k, và n_o sao cho $0 \leq f(n) \leq cg(n)\lg^k(n)$ với tất cả $n \geq n_o\}$.

d. Định nghĩa $\tilde{\Omega}$ và $\tilde{\Theta}$ Theo cách tương tự. Chứng minh sự tương tự tương ứng với Định lý 2.1.

2-6 Các hàm lặp

Có thể áp dụng toán tử lặp "*" dùng trong hàm $\lg^*$ cho các hàm tăng đơn điệu trên các số thực. Với một hàm f thỏa $f(n) < n$, ta định nghĩa hàm $f^{(i)}$ theo đệ quy cho các số nguyên không âm i bằng

$$f^{(i)}(n) = \begin{cases} f(f^{(i-1)}(n)) & \text{nếu } i > 0, \\ n & \text{nếu } i = 0. \end{cases}$$

Với một hằng đã cho $c \in R$, ta định nghĩa hàm lặp f_c^* bằng

$f_c^*(n) = \min\{i \geq 0 : f^{(i)}(n) : \leq c\}$,

không cần được định nghĩa kỹ trong mọi trường hợp. Nói cách khác, lượng $f_c^*(n)$ là số lượng các ứng dụng lặp của hàm f cần thiết để thu giảm đối số của nó thành c hoặc nhỏ hơn.

Với mỗi hàm $f(n)$ dưới đây và các hằng c, cho một cận càng sát càng tốt trên $f_c^*(n)$.

	$f(n)$	c	$f_c^*(n)$
a.	$\lg n$	1	
b.	$n - 1$	0	
c.	$n/2$	1	
d.	$n/2$	2	
e.	$\sqrt{n}$	2	
f.	$\sqrt{n}$	1	
g.	$n^{1/3}$	2	
h.	$n/\lg n$	2	

Ghi chú Chương

Theo Knuth [121], gốc của hệ ký hiệu O bắt nguồn từ một tài liệu lý thuyết-số [number-theory] của P. Bachmann vào năm 1892. Hệ ký hiệu Ω được E. Landau phát minh vào năm 1909 trong tài liệu nói về sự phân phối các số nguyên tố. Các hệ ký hiệu Ω và Θ được Knuth [124] ủng hộ để chỉnh lại cách thực thi tuy phổ dụng song cẩu thả về mặt kỹ thuật của hệ ký hiệu O cho cả cận trên lẫn cận dưới. Nhiều người tiếp tục sử dụng hệ ký hiệu O ở đó hệ ký hiệu Θ chính xác hơn về kỹ thuật. Để tìm hiểu thêm về lịch sử và sự phát triển của các hệ ký hiệu tiệm cận, bạn có thể tìm trong Knuth [121, 124] và Brassard và Bratley [33].

Không phải tất cả tác giả đều định nghĩa các hệ ký hiệu tiệm cận giống như nhau, mặc dù các phần định nghĩa khác nhau cũng ăn ý trong các tình huống phổ dụng nhất. Có vài định nghĩa khác chứa đựng các hàm không mang tính không âm theo tiệm cận, miễn là các giá trị tuyệt đối của chúng được định cận thích hợp.

Để tìm hiểu các tính chất khác của các hàm toán học sơ đẳng, bạn có thể tìm đọc bất kỳ tài liệu tham khảo toán học nào, như Abramowitz và Stegun [1] hoặc Beyer [27], hoặc một cuốn sách về hệ tính toán, như Apostol [12] hay Thomas và Finney [192]. Knuth [121] có đề cập nhiều về toán rời rạc theo nội dung dùng trong khoa học máy tính.

3 Phép Lấy Tổng

Khi một thuật toán chứa cấu trúc điều khiển lặp như vòng lặp **while** hoặc **for**, thời gian thực hiện của nó có thể được diễn tả dưới dạng tổng của các lần bỏ ra cho mỗi đợt thi hành của thân vòng lặp. Ví dụ, trong Đoạn 1.2 ta thấy lần lặp lại thứ j của tiến trình sắp xếp chèn chiếm một thời gian tỷ lệ với j trong ca xấu nhất. Qua việc cộng dồn thời gian bỏ ra cho mỗi lần lặp lại, ta có phép lấy tổng [summation] (hoặc chuỗi)

$$\sum_{j=2}^{n} j.$$

Đánh giá phép lấy tổng này đã cho ra một cận của $\Theta(n^2)$ dựa trên thời gian thực hiện trường hợp xấu nhất của thuật toán. Ví dụ này nêu rõ tầm quan trọng chung của việc hiểu rõ cách điều tác và định cận các phép lấy tổng. (Như sẽ thấy trong Chương 4, các phép lấy tổng cũng nảy sinh khi ta dùng một số phương pháp nhất định để giải quyết các phép truy toán.)

Đoạn 3.1 liệt kê vài công thức căn bản liên quan đến các phép lấy tổng. Đoạn 3.2 cung cấp các kỹ thuật hữu ích để định cận các phép lấy tổng. Các công thức trong Đoạn 3.1 được nêu mà không có chứng minh, tuy Đoạn 3.2 có trình bày các chứng minh của một số công thức này để minh họa các phương pháp trong đoạn đó. Các tài liệu giáo khoa về hệ tính toán đều có đề cập đến hầu hết các phép chứng minh khác.

3.1 Các tính chất và công thức lấy tổng

Cho một dãy $a_1, a_2, \ldots$ con số, tổng hữu hạn $a_1 + a_2 + \ldots + a_n$ có thể viết là

$$\sum_{k=1}^{n} a_k.$$

Nếu $n = 0$, giá trị của phép lấy tổng được định nghĩa là 0. Nếu n không phải là một số nguyên, ta mặc nhận giới hạn trên là $\lfloor n \rfloor$. Cũng vậy, nếu tổng bắt đầu bằng $k = x$, ở đó x không phải là một số nguyên, ta mặc nhận giá trị ban đầu của phép lấy tổng là $\lfloor x \rfloor$. (Nói chung, ta sẽ

đưa vào cận dưới và cận trên một cách tường minh.) Giá trị của một chuỗi hữu hạn luôn được định nghĩa kỹ, và các số hạng của nó có thể được cộng theo một thứ tự bất kỳ.

Cho một dãy số $a_1, a_2, \ldots$, tổng vô hạn $a_1 + a_2 + \ldots$ có thể được viết là

$$\sum_{k=1}^{\infty} a_k.$$

được diễn dịch thành

$$\lim_{n\to\infty} \sum_{k=1}^{n} a_k.$$

Nếu giới hạn không tồn tại, chuỗi sẽ ***phân kỳ;*** bằng không, nó sẽ ***hội tụ***. Không phải lúc nào cũng có thể cộng các số hạng của một chuỗi hội tụ theo một thứ tự bất kỳ. Tuy nhiên, ta có thể dàn xếp lại các số hạng của một ***chuỗi hội tụ tuyệt đối***, nghĩa là, một chuỗi $\sum_{k=1}^{\infty} a_k$ mà chuỗi $\sum_{k=1}^{\infty} |a_k|$ cũng hội tụ.

Tính chất tuyến tính

Với bất kỳ số thực c và bất kỳ dãy hữu hạn $a_1, a_2, \ldots a_n$ và $b_1, b_2, \ldots, b_n$,

$$\sum_{k=1}^{n} (ca_k + b_k) = c\sum_{k=1}^{n} a_k + \sum_{k=1}^{n} b_k.$$

Tính chất tuyến tính cũng được chuỗi hội tụ vô hạn tuân thủ.

Tính chất tuyến tính có thể được khai thác để điều tác các phép lấy tổng liên hợp với hệ ký hiệu tiệm cận. Ví dụ,

$$\sum_{k=1}^{n} \Theta(f(k)) = \Theta\left(\sum_{k=1}^{n} f(k)\right).$$

Trong phương trình này, hệ ký hiệu Θ bên phía trái áp dụng cho biến k, nhưng bên phía phải, nó áp dụng cho n. các dạng điều tác như vậy cũng có thể áp dụng cho chuỗi hội tụ vô hạn.

Chuỗi cấp số cộng

Ta đã sử dụng phép lấy tổng

$$\sum_{k=1}^{n} k = 1 + 2 + \ldots + n,$$

khi phân tích kỹ thuật sắp xếp chèn. Nó là một ***chuỗi cấp số cộng*** [arithmetic series] và có giá trị

$$\sum_{k=1}^{n} k = \frac{1}{2}n(n+1) \qquad (3.1)$$

$$= \Theta(n^2) . \qquad (3.2)$$

Chuỗi cấp số nhân

Với số thực $x \neq 1$, phép lấy tổng

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 + ... + x^n$$

là một ***chuỗi cấp số nhân*** hoặc ***chuỗi lũy thừa*** và có giá trị

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1} . \qquad (3.3)$$

Khi phép lấy tổng là vô hạn và $|x| < 1$, ta có chuỗi cấp số nhân giảm vô hạn

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1 - x} \qquad (3.4)$$

Chuỗi điều hòa

Với các số nguyên dương n, ***số điều hòa*** [harmonic number] thứ n là

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + ... + \frac{1}{n} \qquad (3.5)$$

$$= \sum_{k=1}^{n} \frac{1}{k}$$

$$= \ln n + O(1).$$

Lấy tích phân và lấy vi phân chuỗi

Để tạo thêm các công thức bổ sung, ta lấy tích phân hoặc lấy vi phân các công thức trên đây. Ví dụ, bằng cách lấy vi phân cả hai phía của một chuỗi cấp số nhân vô hạn (3.4) và nhân với x, ta có

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1 - x)^2} \qquad (3.6)$$

Lồng gọn chuỗi

Với bất kỳ dãy $a_0, a_1, ... a_n$

$$\sum_{k=0}^{n-1} (a_k - a_{k-1}) = a_n - a_0, \qquad (3.7)$$

bởi từng số hạng $a_1, a_2, ..., a_{n-1}$ được cộng chính xác một lần và trừ đi

chính xác một lần. Ta nói rằng tổng ***lồng gọn*** [telescopes]. Cũng vậy,

$$\sum_{k=0}^{n-1}(a_k - a_{k+1}) = a_0 - a_n.$$

Để lấy ví dụ về một tổng lồng gọn [telescoping sum], ta xem xét chuỗi

$$\sum_{k=1}^{n-1} = \frac{1}{k(k+1)}.$$

Do có thể viết lại từng số hạng dưới dạng

$$\frac{1}{k(k+1)} = \frac{1}{k} - \frac{1}{(k+1)},$$

ta có

$$\sum_{k=1}^{n-1} \frac{1}{k(k+1)} = \sum_{k=1}^{n-1}\left(\frac{1}{k} - \frac{1}{(k+1)}\right)$$
$$= 1 - \frac{1}{n}.$$

Các Tích

Tích hữu hạn [finite product] $a_1 a_2 ... a_n$ có thể được viết

$$\prod_{k=1}^{n} a_k.$$

Nếu $n = 0$, giá trị của tích được định nghĩa là 1. Để chuyển đổi một công thức có một tích thành một công thức có một phép lấy tổng, ta dùng đồng nhất thức

$$\lg\left(\prod_{k=1}^{n}\right) a_k = \sum_{k=1}^{n} \lg a_k.$$

Bài tập

3.1-1

Tìm một công thức đơn giản cho $\sum_{k=1}^{n}(2k-1)$.

3.1-2 *

Chứng tỏ $\sum_{k=1}^{n} 1/(2k-1) = \ln(\sqrt{n}) + O(1)$ bằng cách điều tác chuỗi điều hòa.

3.1-3 *

Chứng tỏ $\sum_{k=0}^{\infty}(k-1)/2^k = 0$.

3.1-4 *

Đánh giá tổng $\sum_{k=0}^{\infty}(2k+1)x^{2k}$.

3.1-5

Dùng tính chất tuyến tính của các phép lấy tổng để chứng minh rằng $\sum_{k=1}^{n} O(f_k(n)) = O(\sum_{k=1}^{n} f_k(n))$.

3.1-6

Chứng minh $\sum_{k=1}^{\infty} \Omega(f(k)) = \Omega(\sum_{k=1}^{\infty} f(k))$.

3.1-7

Đánh giá tích $\prod_{k=1}^{n} 2.4^k$.

3.1-8 *

Đánh giá tích $\prod_{k=2}^{n} (1 - 1/k^2)$.

3.2 Định cận các phép lấy tổng

Có nhiều kỹ thuật sẵn dùng để định cận [bound] các phép lấy tổng mô tả thời gian thực hiện của các thuật toán. Dưới đây là vài phương pháp phổ dụng nhất.

Phương pháp quy nạp toán học

Cách căn bản nhất để đánh giá một chuỗi đó là dùng phép quy nạp toán học. Để lấy ví dụ, ta hãy chứng minh rằng chuỗi cấp số cộng $\sum_{k=1}^{n} k$ đánh giá theo $\frac{1}{2}n(n+1)$. Ta có thể dễ dàng xác minh điều này với $n = 1$, do đó ta lập giả thiết quy nạp mà nó áp dụng cho n và chứng minh rằng nó áp dụng cho $n + 1$. Ta có

$$\sum_{k=1}^{n-1} k = \sum_{k=1}^{n-1} k + (n+1)$$
$$= \frac{1}{2} n(n+1) + (n+1)$$
$$= \frac{1}{2} n(n+1) + (n+2).$$

Chẳng cần suy đoán giá trị chính xác của một phép lấy tổng để sử dụng phép quy nạp toán học. Phương pháp quy nạp cũng có thể được dùng để nêu một cận. Để lấy ví dụ, hãy chứng minh chuỗi cấp số nhân $\sum_{k=0}^{n} 3^k$ là $O(3^n)$. Cụ thể hơn, hãy chứng minh $\sum_{k=0}^{n} 3^k \le c3^n$ với một hằng c nào đó. Với điều kiện ban đầu $n = 0$, ta có $\sum_{k=0}^{n} 3^k = 1 \le c.1$ miễn là $c \ge 1$. Giả sử, cận áp dụng cho n, hãy chứng minh nó áp dụng cho $n + 1$. Ta có

$$\sum_{k=0}^{n+1} 3^k = \sum_{k=0}^{n} 3^k + 3^{n+1}$$

$$\le c3^n + 3^{n+1}$$
$$= \left(\frac{1}{3} + \frac{1}{c}\right) c3^{n+1}$$
$$\le c3^{n+1}$$

miễn là $(1/3 + 1/c) \le 1$ hoặc, tương đương, $c \ge 3/2$. Như vậy, $\sum_{k=0}^{n} 3^k = O(3^n)$, như ta muốn nêu.

Phải cẩn thận khi dùng hệ ký hiệu tiệm cận để chứng minh các cận bằng phương pháp quy nạp. Xem xét phép chứng minh sai lệch dưới đây rằng $\sum_{k=1}^{n} k = O(n)$. Tất nhiên, $\sum_{k=1}^{1} k = O(1)$. Giả sử cận cho n, giờ đây ta chứng minh nó cho $n+1$:

$$\sum_{k=1}^{n+1} k = \sum_{k=1}^{n} k + (n+1)$$
$$= O(n) + (n+1) \qquad \Leftarrow \textit{sai!!}$$
$$= O(n+1).$$

Mối rối trong biện luận đó là "hằng" được "o-lớn" che giấu sẽ tăng trưởng với n và do đó không bất biến. Ta đã không chứng tỏ rằng cùng một hằng làm việc với *tất cả mọi n*.

Định cận các số hạng

Đôi lúc, để có được một cận trên tốt trên một chuỗi, ta có thể định cận từng số hạng của chuỗi, và thông thường sử dụng số hạng lớn nhất là đủ để định cận các số hạng khác. Ví dụ, một cận trên nhanh trên chuỗi cấp số cộng (3.1) là

$$\sum_{k=1}^{n} k \le \sum_{k=1}^{n} n$$
$$= n^2 .$$

Nói chung, với một chuỗi $\sum_{k=1}^{n} a_k$. nếu để $a_{max} = \max_{1 \le k \le n} a_k$, ta có

$$\sum_{k=1}^{n} a_k \le n a_{max}.$$

Kỹ thuật định cận từng số hạng trong một chuỗi bằng số hạng lớn nhất là một phương pháp yếu khi thực tế chuỗi có thể được định cận bằng chuỗi cấp số nhân. Cho chuỗi $\sum_{k=0}^{n} a_k$, giả sử rằng $a_{k+1}/a_k \le r$ với tất cả $k \ge 0$, ở đó $r < 1$ là một hằng. Tổng có thể được định cận bằng một chuỗi cấp số nhân giảm vô hạn, bởi $a_k \le a_0 r^k$, và như vậy

$$\sum_{k=0}^{n} a_k \le \sum_{k=0}^{\infty} a_0 r^k$$
$$= a_0 \sum_{k=0}^{\infty} r^k$$
$$= a_0 \frac{1}{1-r} .$$

Ta có thể áp dụng phương pháp này để định cận phép lấy tổng $\sum_{k=1}^{\infty}(k/3^k)$. Số hạng đầu tiên là 1/3, và tỷ số của các số hạng liên tiếp là

$$\frac{(k+1)/3^{k+1}}{k/3^k} = \frac{1}{3}\cdot\frac{k+1}{k} \le \frac{2}{3}$$

với tất cả $k \ge 1$. Do đó, mỗi số hạng được định cận trên bằng $(1/3)(2/3)^k$, sao cho

$$\sum_{k=1}^{\infty}\frac{2}{3^k} \le \sum_{k=1}^{\infty}\frac{1}{3^k}\left(\frac{2}{3}\right) = \frac{1}{3}\cdot\frac{1}{1-2/3} = 1.$$

Một mối rối thường gặp khi áp dụng phương pháp này đó là chứng tỏ tỷ số của các số hạng liên tiếp nhỏ hơn 1 rồi mặc nhận rằng phép lấy tổng được định cận bằng của chuỗi cấp số nhân. Một ví dụ đó là chuỗi điều hòa vô hạn [infinite harmonic series], nó phân kỳ bởi

$$\sum_{k=1}^{\infty}\frac{1}{k} = \lim_{n\to\infty}\sum_{k=1}^{n}\frac{1}{k} = \lim_{n\to\infty}\Theta(\lg n) = \infty.$$

Tỷ số của các số hiệu thứ $(k+1)$ và k trong chuỗi này là $k/(k+1) < 1$, nhưng chuỗi không được định cận bởi một chuỗi cấp số nhân giảm. Để định cận một chuỗi bằng một chuỗi cấp số nhân, ta phải chứng tỏ tỷ số được định cận tách ra khỏi 1; nghĩa là, phải có một $r < 1$, là một *hằng*, sao cho tỷ số của tất cả các cặp số hạng liên tiếp không bao giờ vượt quá r. Trong chuỗi điều hòa, không có dạng r như vậy tồn tại bởi tỷ số trở nên tùy ý sát với 1.

Tách các phép lấy tổng

Một cách để có các cận dựa trên một phép lấy tổng khác đó là diễn tả chuỗi dưới dạng tổng của hai hay nhiều chuỗi bằng cách phân hoạch [partitioning] miền của chỉ số rồi định cận từng chuỗi kết quả. Ví dụ, giả sử ta gắng tìm một cận dưới dựa trên chuỗi cấp số cộng $\sum_{k=1}^{n} k$, đã được chứng tỏ là có một cận trên n^2. Ta có thể gắng định cận từng số hạng trong phép lấy tổng bằng số hạng nhỏ nhất, nhưng do số hạng đó là 1, ta có một cận dưới n của phép lấy tổng—cách xa với cận trên n^2 của chúng ta.

Để có một cận dưới tốt hơn, trước tiên ta tách phép lấy tổng. Để tiện dụng, giả sử n là chẵn. Ta có

$$\begin{aligned}\sum_{k=1}^{n} k &= \sum_{k=1}^{n/2} k + \sum_{k=n/2+1}^{n} k \\ &\geq \sum_{k=1}^{n/2} 0 + \sum_{k=n/2+1}^{n} (n/2) \\ &\geq (n/2)^2 \\ &= \Omega(n^2),\end{aligned}$$

là một cận sát tiệm cận, bởi $\sum_{k=1}^{n} k = O(n^2)$.

Với một phép lấy tổng phát sinh từ việc phân tích một thuật toán, ta thường có thể tách phép lấy tổng và bỏ qua một số lượng bất biến các số hạng ban đầu. Nói chung, kỹ thuật này áp dụng khi từng số hạng a_k trong phép lấy tổng $\sum_{k=0}^{n} a_k$ độc lập với n. Như vậy với bất kỳ bằng $k_0 > 0$, ta có thể viết

$$\begin{aligned}\sum_{k=1}^{n} a_k &= \sum_{k=0}^{k_0-1} a_k + \sum_{k=k_0}^{n} a_k \\ &= \Theta(1) + \sum_{k=k_0}^{n} a_k,\end{aligned}$$

bởi tất cả số hạng ban đầu của phép lấy tổng là bất biến và chúng có một số lượng bất biến. Như vậy, ta có thể dùng các phương pháp khác để định cận $\sum_{k=k_0}^{n} a_k$. Ví dụ, để tìm một cận trên tiệm cận trên

$$\sum_{k=0}^{\infty} \frac{k^2}{2^k},$$

ta nhận thấy tỷ số của các số hạng liên tiếp là

$$\begin{aligned}\frac{(k+1)^2/2^{k+1}}{k^2/2^k} &= \frac{(k+1)^2}{2k^2} \\ &\leq \frac{8}{9}\end{aligned}$$

nếu $k \geq 3$. Như vậy, phép lấy tổng có thể được tách thành

$$\begin{aligned}\sum_{k=0}^{\infty} \frac{k^2}{2^k} &= \sum_{k=1}^{\infty} \frac{k^2}{2^k} + \sum_{k=3}^{\infty} \frac{k^2}{2^k} \\ &\leq O(1) + \frac{9}{8} \sum_{k=0}^{\infty} \left(\frac{8}{9}\right)^k \\ &= O(1),\end{aligned}$$

bởi phép lấy tổng thứ hai là một chuỗi cấp số nhân giảm.

Kỹ thuật tách các phép lấy tổng có thể được dùng để xác định các

cận tiệm cận trong các tình huống khó khăn hơn nhiều. Ví dụ, có thể đạt được một cận $O(\lg n)$ trên chuỗi điều hòa (3.5):

$$H_n = \sum_{k=1}^{n} \frac{1}{k} .$$

Ý tưởng đó là tách miền 1 đến n thànhh các mẩu $\lfloor \lg n \rfloor$ và định cận trên phần đóng góp của mỗi mẩu theo 1. Như vậy,

$$\begin{aligned}
\sum_{k=1}^{n} \frac{1}{k} &\le \sum_{i=0}^{\lfloor \lg n \rfloor} \sum_{j=0}^{2^i-1} \frac{1}{2^i + j} \\
&\le \sum_{i=0}^{\lfloor \lg n \rfloor} \sum_{j=0}^{2^i-1} \frac{1}{2^i} \\
&\le \sum_{i=0}^{\lfloor \lg n \rfloor} 1 \\
&\le \lg n + 1 . \qquad (3.8)
\end{aligned}$$

Phép xấp xỉ bằng các tích phân

Khi một phép lấy tổng có thể được diễn tả dưới dạng $\sum_{k=m}^{n} f(k)$, ở đó $f(k)$ là một hàm tăng đơn điệu, ta có thể lấy xấp xỉ nó bằng các tích phân:

$$\int_{m-1}^{n} f(x)dx \le \sum_{k=m}^{n} f(k) \le \int_{m}^{n+1} f(x)dx. \qquad (3.9)$$

Hình 3.1 chứng minh phép xấp xỉ này. Phép lấy tổng được biểu thị bằng diện tích các hình chữ nhật trong hình, và tích phân là vùng tô bóng dưới đường cung. Khi $f(k)$ là một hàm giảm đơn điệu, ta có thể dùng phương pháp tương tự để cung cấp các cận

$$\int_{m}^{n+1} f(x)dx \le \sum_{k=m}^{n} f(k) \le \int_{m-1}^{n} f(x)dx. \qquad (3.10)$$

Phép xấp xỉ tích phân (3.10) cho ra một ước tính chặt cho số điều hòa thứ n. Với một cận dưới, ta có

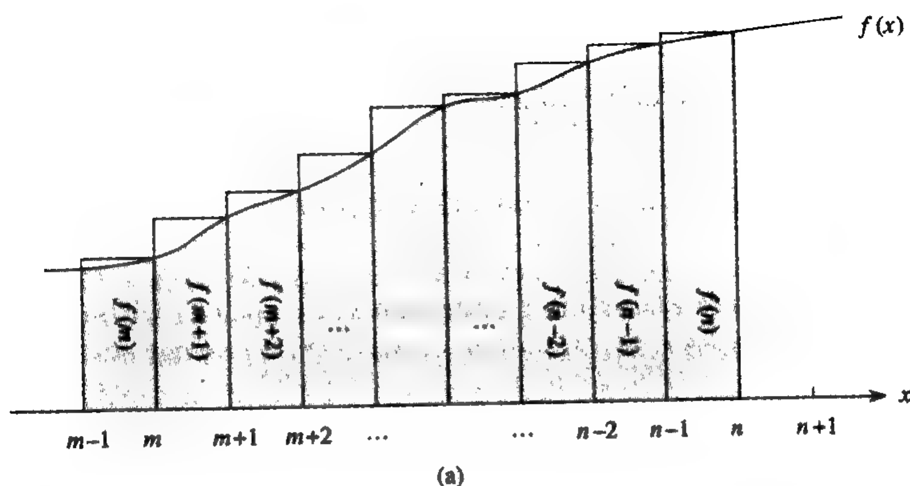
$$\begin{aligned}
\sum_{k=1}^{n} \frac{1}{k} &\ge \int_{1}^{n+1} \frac{dx}{x} \\
&= \ln(n+1). \qquad (3.11)
\end{aligned}$$

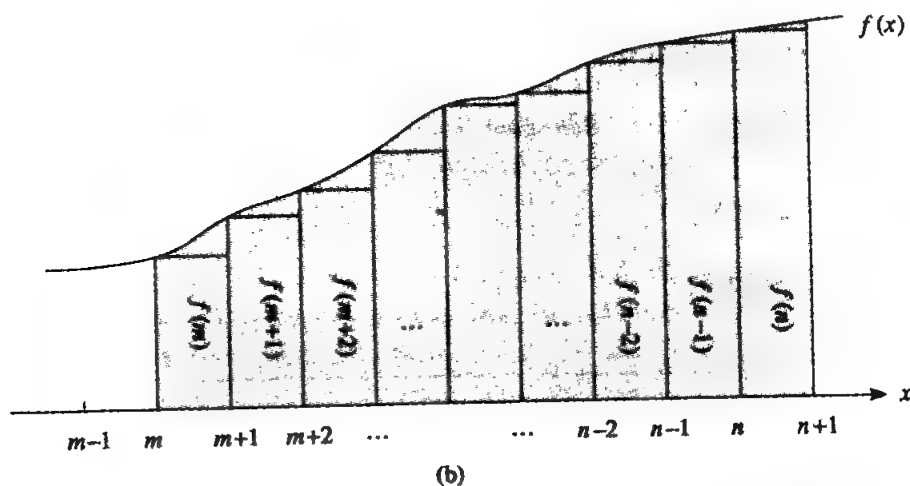
Với một cận trên, ta rút ra bất đẳng thức

$$\begin{aligned}
\sum_{k=2}^{n} \frac{1}{k} &\le \int_{1}^{n} \frac{dx}{x} \\
&= \ln n,
\end{aligned}$$

cho ra cận

$$\sum_{k=1}^{n} \frac{1}{k} \le \ln n + 1. \qquad (3.12)$$





Hình 3.1 Phép xấp xỉ của $\sum_{k=m}^{n} f(k)$, bằng các tích phân. Diện tích của từng hình chữ nhật được nêu trong hình chữ nhật, và tổng diện tích chữ nhật biểu thị cho giá trị của phép lấy tổng. Tích phân được biểu thị bằng diện tích tô bóng dưới đường cung. Bằng cách so sánh các diện tích trong (**a**), ta có $\int_{m-1}^{n} f(x)dx \le \sum_{k=m}^{n} f(k)$, rồi bằng cách chuyển các hình chữ nhật sang phải một đơn vị, ta có $\sum_{k=m}^{n} f(k) \le \int_{m-1}^{n+1} f(x)dx$ trong (**b**).

Bài tập

3.2-1

Chứng tỏ $\sum_{k=1}^{n} 1/k^2$ được định cận bên trên bằng một hằng.

3.2-2

Tìm một cận trên tiệm cận theo phép lấy tổng

$$\sum_{k=0}^{\lfloor \lg n \rfloor} \lceil n/2^k \rceil .$$

3.2-3

Chứng tỏ số điều hòa thứ n là $\Omega(\lg n)$ bằng cách tách phép lấy tổng.

3.2-4

Tính xấp xỉ $\sum_{k=1}^{n} k^3$ bằng một tích phân.

3.2-5

Tại sao ta không dùng phép xấp xỉ tích phân (3.10) trực tiếp trên $\sum_{k=1}^{n} 1/k$ để có một cận trên dựa trên số điều hòa thứ n?

Các Bài Toán

3-1 Định cận các phép lấy tổng

Nêu các cận sát tiệm cận dựa trên các phép lấy tổng dưới đây. Giả sử $r \geq 0$ và $s \geq 0$ là các hằng.

a. $\sum_{k=1}^{n} k^r$.

b. $\sum_{k=1}^{n} \lg^s k$.

c. $\sum_{k=1}^{n} k^r \lg^s k$.

Ghi chú Chương

Knuth [121] là tài liệu tham khảo tuyệt vời cho nội dung trình bày trong chương này. để tìm hiểu các tính chất căn bản của chuỗi, bạn có thể xem bất kỳ cuốn sách nào về hệ tính toán, như Apostol [12] hoặc Thomas and Finney [192].

4 Các Phép Truy Toán

Như đã lưu ý trong Chương 1, khi một thuật toán chứa một lệnh gọi đệ quy lên chính nó, ta có thể dùng phép truy toán để mô tả thời gian thực hiện của nó. Một ***phép truy toán*** là một phương trình hoặc bất đẳng thức mô tả một hàm theo dạng giá trị của nó dựa trên các nhập liệu nhỏ hơn. Ví dụ, ta đã thấy trong Chương 1 rằng thời gian thực hiện trường hợp xấu nhất $T(n)$ của thủ tục MERGE-SORT có thể được mô tả bằng phép truy toán

$$T(n) = \begin{cases} \Theta(1) & \text{nếu } n = 1, \\ 2T(n/2) + \Theta(n) & \text{nếu } n > 1, \end{cases} \qquad (4.1)$$

mà nghiệm của nó được xác nhận là $T(n) = \Theta(n\lg n)$.

Chương này cung cấp ba phương pháp để giải quyết các phép truy toán—nghĩa là, để có được các cận tiệm cận "Θ" hoặc "O" dựa trên nghiệm. Trong ***phương pháp thay thế***, ta suy đoán một cận rồi dùng phép quy nạp toán học để chứng minh sự suy đoán của ta là đúng. ***Phương pháp lặp*** chuyển đổi phép truy toán thành một phép lấy tổng rồi dựa trên các kỹ thuật định cận các phép lấy tổng để giải quyết phép truy toán. ***Phương pháp chủ*** cung cấp các biên cho các phép truy toán theo dạng

$$T(n) = aT(n/b) + f(n),$$

ở đó $a \geq 1$, $b > 1$, và $f(n)$ là một hàm đã cho; ta cần nhớ ba trường hợp này; và một khi đã nhớ, việc xác định các cận tiệm cận cho nhiều phép truy toán đơn giản sẽ trở nên dễ dàng.

Tiểu tiết kỹ thuật

Trong thực tế, ta không để ý đến một số chi tiết kỹ thuật nhất định khi phát biểu và giải quyết các phép truy toán. Một ví dụ tốt về một chi tiết thường được xem nhẹ đó là giả thiết về các đối số số nguyên cho các hàm. Thông thường, thời gian thực hiện $T(n)$ của một thuật toán chỉ được định nghĩa khi n là một số nguyên, bởi với hầu hết các thuật toán, kích cỡ nhập liệu luôn là một số nguyên. Ví dụ, phép truy toán mô tả thời gian thực hiện trường hợp xấu nhất của MERGE-SORT thực sự là

$$T(n) = \begin{cases} \Theta(1) & \text{nếu } n = 1, \\ 2T(\lceil n/2 \rceil) + 2T(\lfloor n/2 \rfloor) + \Theta(n) & \text{nếu } n > 1, \end{cases} \quad (4.2)$$

Các ***điều kiện biên*** [boundary conditions] biểu thị cho một lớp chi tiết khác mà ta thường bỏ qua. Bởi thời gian thực hiện của một thuật toán trên một nhập liệu có kích cỡ bất biến là một hằng, các phép truy toán nảy sinh từ các thời gian thực hiện của các thuật toán thường có $T(n) = \Theta(1)$ với n đủ nhỏ. Kết quả là, để tiện dụng, ta thường bỏ qua các phát biểu về các điều kiện biên của các phép truy toán và mặc nhận rằng $T(n)$ là bất biến với n nhỏ. Ví dụ, ta thường phát biểu phép truy toán (4.1) dưới dạng

$$T(n) = 2T(n/2) + \Theta(n), \quad (4.3)$$

mà không rõ rệt gán các giá trị cho n nhỏ. Sở dĩ như vậy là vì mặc dù việc thay đổi giá trị của $T(1)$ sẽ làm thay đổi nghiệm đối với phép truy toán, song nghiệm thường không thay đổi theo nhiều hơn một thừa số bất biến, do đó cấp tăng trưởng là không thay đổi.

Khi phát biểu và giải quyết các phép truy toán, ta thường bỏ qua cận dưới, cận trên, và các điều kiện biên. Ta tiến tới mà không cần các tiểu tiết này và về sau xác định xem chúng có ý nghĩa hay không. Thường thì không, song điều quan trọng là phải biết khi nào chúng có ý nghĩa. Kinh nghiệm sẽ giúp ta điều đó, và vì thế một số định lý cũng phát biểu rằng các chi tiết này không ảnh hưởng đến các cận tiệm cận của nhiều phép truy toán mà ta gặp trong khi phân tích các thuật toán (xem Định lý 4.1 và Bài toán 4-5). Tuy nhiên, trong chương này, ta sẽ đề cập vài chi tiết đó để cho thấy các điểm tế nhị của các phương pháp giải quyết phép truy toán.

4.1 Phương pháp thay thế

Phương pháp thay thế để giải quyết các phép truy toán thường liên quan đến việc suy đoán dạng thức của nghiệm rồi dùng phép quy nạp toán học để tìm ra các hằng và chứng tỏ nghiệm làm việc. Tên gọi xuất xứ từ việc dùng đáp án suy đoán thay cho hàm khi giả thuyết quy nạp được áp dụng cho các giá trị nhỏ hơn. Tuy phương pháp này mạnh, song hiển nhiên chỉ có thể áp dụng nó trong các trường hợp dễ dàng suy đoán dạng thức của đáp án.

Có thể dùng phương pháp thay thế để thiết lập các cận trên hoặc cận dưới cho một phép truy toán. Để lấy ví dụ, hãy xác định một cận trên cho phép truy toán

$$T(n) = 2T(\lfloor n/2 \rfloor) + n, \quad (4.4)$$

nó tương tự như các phép truy toán (4.2) và (4.3). Ta suy đoán nghiệm là $T(n) = O(n \lg n)$. Phương pháp của chúng ta là chứng minh rằng $T(n) \le cn\lg n$ với một chọn lựa thích hợp cho hằng $c > 0$. Để bắt đầu, ta giả định cận này áp dụng cho $\lfloor n/2 \rfloor$, nghĩa là, $T(\lfloor n/2 \rfloor) \le c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$. Thay vào phép truy toán cho ra

$$\begin{aligned} T(n) &\le 2(c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \\ &\le cn \lg(n/2) + n \\ &= cn\lg n - cn\lg 2 + n \\ &= cn\lg n - cn + n \\ &\le cn\lg n , \end{aligned}$$

ở đó bước cuối có thể áp dụng miễn là $c \ge 1$.

Phương pháp quy nạp toán học giờ đây yêu cầu ta chứng tỏ nghiệm áp dụng cho các điều kiện biên. Nghĩa là, ta phải chứng tỏ có thể chọn hằng c đủ lớn sao cho cận $T(n) \le cn \lg n$ cũng áp dụng cho các điều kiện biên. Đôi lúc yêu cầu này có thể dẫn đến các vấn đề. Để biện luận, ta giả sử $T(1) = 1$ là điều kiện biên duy nhất của phép truy toán. Đáng tiếc, như vậy ta không thể chọn c đủ lớn, bởi $T(1) \le c\, 1 \lg 1 = 0$.

Có thể dễ dàng khắc phục khó khăn trong việc chứng minh một giả thiết quy nạp cho một điều kiện biên cụ thể. Ta vận dụng sự việc hệ ký hiệu tiệm cận chỉ yêu cầu chứng minh $T(n) \le cn \lg n$ với $n \ge n_0$, ở đó n_0 là một hằng. Điểm mấu chốt đó là không xem xét điều kiện biên khó $T(1) = 1$ trong phép chứng minh quy nạp và gộp $n = 2$ và $n = 3$ dưới dạng một phần của các điều kiện biên cho phép chứng minh. Ta có thể áp đặt $T(2)$ và $T(3)$ làm các điều kiện biên cho phép chứng minh quy nạp bởi với $n > 3$, phép truy toán không trực tiếp lệ thuộc vào $T(1)$. Từ phép truy toán, ta suy ra $T(2) = 4$ và $T(3) = 5$. Giờ đây để hoàn tất phép chứng minh quy nạp $T(n) \le cn \lg n$ với một $c \ge 2$ nào đó, ta chọn c đủ lớn sao cho $T(2) \le c2 \lg 2$ và $T(3) \le c3 \lg 3$. Và như vậy, một chọn lựa $c \ge 2$ bất kỳ đều đủ. Với hầu hết các phép truy toán sẽ được xem xét ở đây, ta chỉ việc mở rộng các điều kiện biên để khiến giả thiết quy nạp làm việc với n nhỏ.

Để suy đoán tốt

Đáng tiếc, không có một cách chung nào để suy đoán các nghiệm đúng đắn cho các phép truy toán. Việc suy đoán một nghiệm đòi hỏi sự kinh nghiệm và, đôi lúc, cả óc sáng tạo. Tuy nhiên, may thay, có vài phép phỏng đoán [heuristics] có thể giúp ta trở thành người suy đoán tốt.

Nếu một phép truy toán tương tự như đã gặp trước đó, thì việc suy đoán một nghiệm tương tự là hợp lý. Để lấy ví dụ, ta xem xét phép truy toán

$T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$,

nó có vẻ khó bởi "17" trong đối số được cộng vào T bên phía phải. Tuy nhiên, theo trực giác, số hạng bổ sung này thực chất không thể ảnh hưởng đến nghiệm đối với phép truy toán. Khi n lớn, sự khác biệt giữa $T(\lfloor n/2 \rfloor)$ và $T(\lfloor n/2 \rfloor + 17)$ không lớn đến như vậy: cả hai đều cắt đôi n gần đều nhau. Kết quả, ta đưa ra suy đoán rằng $T(n) = O(n\lg n)$, mà bạn có thể xác minh là đúng bằng phương pháp thay thế (xem Bài tập 4.1-5).

Một cách khác để đưa ra một suy đoán tốt đó là chứng minh các cận trên và cận dưới "lơi" [loose] trên phép truy toán rồi thu nhỏ miền bất định. Ví dụ, có thể bắt đầu bằng một cận dưới $T(n) = \Omega(n)$ cho phép truy toán (4.4), bởi ta có số hạng n trong phép truy toán, và có thể chứng minh cận trên ban đầu là $T(n) = O(n^2)$. Như vậy, có thể từ từ hạ thấp cận trên và nâng cận dưới lên cho đến khi hội tụ trên nghiệm đúng, sát theo tiệm cận là $T(n) = \Theta(n\lg n)$.

Các điểm tinh tế

Có những lúc ở đó có thể suy đoán đúng đắn tại một cận tiệm cận cho nghiệm của một phép truy toán, nhưng không hiểu làm sao toán học dường như có gì trục trặc trong phương pháp quy nạp. Thông thường, vấn đề đó là giả thiết quy nạp không đủ mạnh để chứng minh cận chi tiết [detailed bound]. Khi gặp phải một dạng khó khăn đột xuất như vậy, việc xem lại phép suy đoán bằng cách trừ một số hạng cấp thấp thường cho phép khắc phục vấn đề toán học.

Hãy xem xét phép truy toán

$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$.

Ta suy đoán rằng nghiệm là $O(n)$, và gắng chứng tỏ $T(n) \leq cn$ với một chọn lựa thích hợp cho hằng c. Thay thế suy đoán trong phép truy toán, ta có

$$\begin{aligned} T(n) \lfloor n/2 \rfloor &\leq c \lfloor n/2 \rfloor + c \lceil n/2 \rceil + 1 \\ &= cn + 1 , \end{aligned}$$

điều này không hàm ý $T(n) \leq cn$ với bất kỳ chọn lựa nào của c. Nó đang lôi kéo ta thử một suy đoán lớn hơn, giả sử $T(n) = O(n^2)$, có thể được tạo để làm việc, nhưng trên thực tế, suy đoán của ta đối với nghiệm $T(n) = O(n)$ là đúng đắn. Tuy nhiên, để chứng tỏ điều này, ta phải tạo một giả thuyết quy nạp mạnh hơn.

Về trực giác, suy đoán của ta là gần đúng: chỉ để hụt hằng 1, một số hạng cấp thấp. Tuy vậy, phép quy nạp toán học sẽ không làm việc trừ phi ta chứng minh dạng thức chính xác của giả thuyết quy nạp. Để khắc

phục khó khăn này, ta *trừ* một số hạng cấp thấp ra khỏi suy đoán trên đây. Suy đoán mới sẽ là $T(n) \le cn - b$, ở đó $b \ge 0$ là bất biến. Giờ đây ta có

$$\begin{aligned} T(n) &\le (c\lfloor n/2 \rfloor - b) + (c(\lceil n/2 \rceil - b) + 1 \\ &= cn - 2b + 1 \\ &\le cn - b , \end{aligned}$$

miễn là $b \ge 1$. Cũng như trước, hằng c phải được chọn đủ lớn để điều quản các điều kiện biên.

Đa số người nhận thấy ý tưởng trừ một số hạng cấp thấp trái với trực giác. Xét cho cùng, nếu toán học không giải được, phải chăng ta không được gia tăng sự suy đoán của mình? Điểm mấu chốt để hiểu bước này là nhớ kỹ ta đang dùng phép quy nạp toán học: có thể chứng minh một nội dung mạnh hơn với một giá trị đã cho bằng cách mặc nhận một nội dung mạnh hơn với các giá trị nhỏ hơn.

Tránh các chỗ bẫy

Sẽ dễ dàng gặp lỗi khi dùng hệ ký hiệu tiệm cận. Ví dụ, trong phép truy toán (4.4) ta có thể chứng minh sai $T(n) = O(n)$ bằng cách suy đoán $T(n) \le cn$

rồi biện luận

$$\begin{aligned} T(n) &\le 2(c\lfloor n/2 \rfloor) + n \\ &\le cn + n \\ &= O(n) , \qquad \Leftarrow \textit{sai !!} \end{aligned}$$

bởi c là một hằng. Lỗi đó là ta chưa chứng minh dạng chính xác của giả thuyết quy nạp, nghĩa là, $T(n) \le cn$.

Thay đổi các biến

Đôi lúc, chỉ cần một điều tác đại số nhỏ là có thể khiến một phép truy toán vô danh trở thành tương tự như đã gặp trên đây. Để lấy ví dụ, hãy xem xét phép truy toán

$$T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \lg n ,$$

Nó có vẻ khó. Tuy nhiên, có thể đơn giản hóa phép truy toán này bằng cách thay đổi các biến. Để tiện dụng, ta sẽ không quan tâm về việc làm tròn các giá trị, như $\sqrt{n}$, sẽ là các số nguyên. Gọi tên lại $m = \lg n$ cho ra

$$T(2^m) = 2T(2^{m/2}) + m .$$

Giờ đây có thể đặt tên lại $S(m) = T(2^m)$ để tạo phép truy toán mới $S(m) = 2S(m/2) + m$,

trông rất giống phép truy toán (4.4) và có cùng nghiệm: $S(m) = O(m \lg m)$. Thay đổi ngược từ $S(m)$ thành $T(n)$, ta có $T(n) \doteq T(2^m) = S(m) = O(m\lg m) = O(\lg n \lg\lg n)$.

Bài tập

4.1-1

Chứng tỏ nghiệm của $T(n) = T(\lceil n/2 \rceil) + 1$ là $O(\lg n)$.

4.1-2

Chứng tỏ nghiệm của $T(n) = 2T(\lfloor n/2 \rfloor) + n$ là $\Omega(n\lg n)$. Kết luận nghiệm là $\Theta(n\lg n)$.

4.1-3

Chứng tỏ bằng cách lập một giả thuyết quy nạp khác, ta có thể khắc phục khó khăn bằng điều kiện biên $T(1) = 1$ với phép truy toán (4.4) mà không điều chỉnh các điều kiện biên với phép chứng minh quy nạp.

4.1-4

Chứng tỏ $\Theta(n \lg n)$ là nghiệm cho phép truy toán "chính xác" (4.2) để sắp xếp trộn.

4.1-5

Chứng tỏ nghiệm cho $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$ là $O(n\lg n)$.

4.1-6

Giải phép truy toán $T(n) = 2T(\sqrt{n}) + 1$ bằng cách thay đổi các biến. Đừng quan tâm về việc các giá trị có phải là tích phân hay không.

4.2 Phương pháp lặp

Phương pháp lặp một phép truy toán không yêu cầu ta suy đoán câu trả lời, nhưng nó có thể yêu cầu nhiều đại số hơn phương pháp thay thế. Ý tưởng đó là mở rộng (lặp lại) phép truy toán và diễn tả nó dưới dạng một phép lấy tổng các số hạng chỉ lệ thuộc vào n và các điều kiện ban đầu. Như vậy, các kỹ thuật đánh giá các phép lấy tổng có thể được dùng để cung cấp các cận dựa trên nghiệm.

Để lấy ví dụ, ta xem xét phép truy toán

$$T(n) = 3T(\lfloor n/4 \rfloor) + n .$$

Ta lập nó như sau:

$$\begin{aligned} T(n) &= n + 3T((\lfloor n/4 \rfloor) \\ &= n + 3((\lfloor n/4 \rfloor + 3T((\lfloor n/16 \rfloor)) \end{aligned}$$

$$= n + 3(\lfloor n/4 \rfloor + 3(\lfloor n/16 \rfloor + 3T(\lfloor n/64 \rfloor)))$$
$$= n + 3\lfloor n/4 \rfloor + 9\lfloor n/16 \rfloor + 27T(\lfloor n/64 \rfloor),$$

ở đó $\lfloor \lfloor n/4 \rfloor /4 \rfloor = \lfloor n/16 \rfloor$ và $\lfloor \lfloor n/16 \rfloor /4 \rfloor = \lfloor n/64 \rfloor$ sinh ra từ đồng nhất thức (2.4).

Phải lặp phép truy toán đến đâu trước khi đạt đến một điều kiện biên? Số hạng thứ i trong chuỗi là $3^i \lfloor n/4^i \rfloor$. Phép lặp đụng $n = 1$ khi $\lfloor n/4^i \rfloor = 1$ hoặc, tương đương, khi i vượt quá $\log_4 n$. Bằng cách tiếp tục lặp lại cho đến điểm này và dùng cận $\lfloor n/4^i \rfloor \leq n/4^i$, ta phát hiện phép lấy tổng chứa một chuỗi cấp số nhân giảm:

$$\begin{aligned} T(n) &\leq n + 3n/4 + 9n/16 + 27n/64 + \dots + 3^{\log_4 n}\Theta(1) \\ &\leq \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i + \Theta(n^{\log_4 3}) \\ &= 4n + o(n) \\ &= O(n). \end{aligned}$$

Ở đây, ta đã dùng đồng nhất thức (2.9) để kết luận rằng $3^{\log_4 n} = n^{\log_4 3}$, và ta đã dùng sự việc $\log_4 3 < 1$ để kết luận $\Theta(n^{\log_4 3}) = o(n)$.

Phương pháp lặp thường dẫn đến khá nhiều đại số, và việc duy trì mọi thứ mạch lạc có thể là một thách thức. Điểm mấu chốt là tập trung vào hai tham số: số lần mà phép truy toán cần được lặp lại để đạt điều kiện biên, và tổng các số hạng nảy sinh từ mỗi cấp của tiến trình lặp lại. Đôi lúc, trong tiến trình lặp lại một phép truy toán, bạn có thể suy đoán nghiệm mà không cần giải quyết toàn bộ toán học. Như vậy, phép lặp có thể được loại bỏ để thay bằng phương pháp thay thế, thường yêu cầu ít đại số hơn.

Khi một phép truy toán chứa các hàm sàn và trần, toán học có thể trở nên đặc biệt phức tạp. Thường, ta nên mặc nhận rằng phép truy toán chỉ được định nghĩa dựa trên các lũy thừa chính xác của một con số. Trong ví dụ của chúng ta, nếu đã mặc nhận $n = 4^k$ với một số nguyên k nào đó, các hàm sàn có thể đã được bỏ qua để tiện dụng. Đáng tiếc, việc chứng minh cận $T(n) = O(n)$ với chỉ các lũy thừa chính xác của 4 xét về mặt kỹ thuật là một lạm dụng của hệ ký hiệu O. Các phần định nghĩa của hệ ký hiệu tiệm cận yêu cầu chứng minh các cận đó với *tất cả các* số nguyên đủ lớn, chứ không chỉ có các lũy thừa của 4. Ta sẽ thấy trong Đoạn 4.3, với một lớp các phép truy toán lớn, chi tiết kỹ thuật này có thể khắc phục được. Bài toán 4-5 cũng đưa ra các điều kiện qua đó một phân tích với các lũy thừa chính xác của một số nguyên có thể mở rộng đến tất cả các số nguyên.

Các cây đệ quy

Cây đệ quy [recursion tree] là cách tiện dụng để hình dung nội dung xảy ra khi một phép truy toán được lặp lại, và nó có thể giúp tổ chức việc theo dõi diễn tiến đại số cần thiết để giải phép truy toán. Nó đặc biệt hữu ích khi phép truy toán mô tả một thuật toán chia để trị. Hình 4.1 có nêu tạo hàm của cây đệ quy với

$T(n) = 2T(n/2) + n^2$.

Để tiện dụng, ta mặc nhận rằng n là một lũy thừa chính xác của 2. Phần (a) trong hình nêu $T(n)$, mà trong phần (b) đã được mở rộng thành một cây tương đương biểu thị phép truy toán. Toán hạng n^2 là gốc (hao phí tại cấp trên cùng của đệ quy), và hai cây con của gốc là hai phép truy toán $T(n/2)$ nhỏ hơn. Phần (c) nêu tiến trình này được tiến hành một bước xa hơn bằng cách mở rộng $T(n/2)$. Hao phí cho mỗi trong hai nút con [subnodes] tại cấp thứ hai của đệ quy là $(n/2)^2$. Ta tiếp tục mở rộng mỗi nút trong cây bằng cách tách thành các thành phần cấu tạo của nó như được xác định bởi phép truy toán, cho đến khi đạt được một điều kiện biên. Phần (d) nêu cây kết quả.

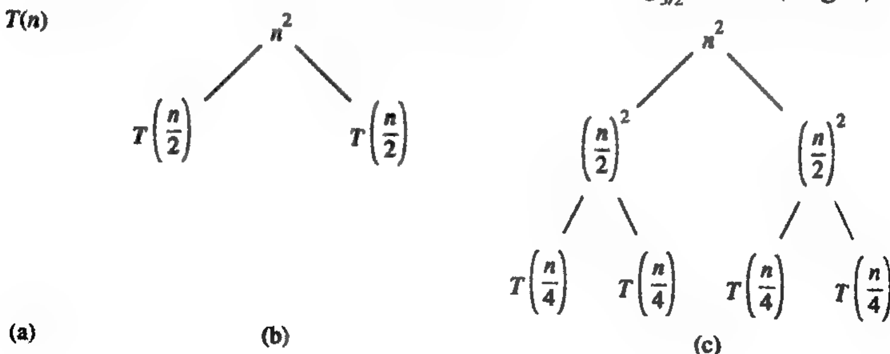
Giờ đây để đánh giá phép truy toán, ta cộng các giá trị qua từng cấp của cây. Cấp trên cùng có tổng giá trị n^2, cấp thứ hai có giá trị $(n/2)^2+(n/2)^2 = n^2/2$, cấp thứ ba có giá trị $(n/4)^2+(n/4)^2+(n/4)^2+ (n/4)^2 = n^2/4$, và vân vân. Bởi các giá trị giảm theo cấp số nhân, nên tổng tối đa là một thừa số bất biến hơn số hạng lớn nhất (đầu tiên), và do đó nghiệm là $\Theta(n^2)$.

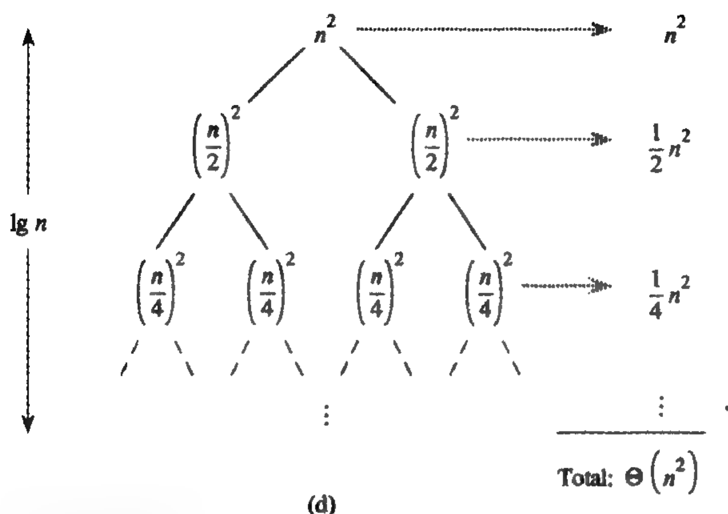
Một ví dụ khác, phức tạp hơn, Hình 4.2 có nêu cây đệ quy của

$T(n) = T(n/3) + T(2n/3) + n$.

(Ở đây cũng vậy, để đơn giản, ta bỏ qua các hàm sàn và trần.) Khi cộng các giá trị qua các cấp của cây đệ quy, ta có một giá trị n cho mọi cấp. Lộ trình dài nhất từ gốc đến một lá là $n \rightarrow (2/3)n \rightarrow (2/3)^2 n \rightarrow \ldots \rightarrow 1$.

Do $(2/3)^k n = 1$ khi $k = \log_{3/2} n$, nên chiều cao của cây là $\log_{3/2} n$. Như vậy, nghiệm cho phép truy toán tối đa là $n \log_{3/2} n = O(n \lg n)$.





Hình 4.1 Kiến tạo của một cây đệ quy cho phép truy toán $T(n) = 2T(n/2) + n^2$. Phần (**a**) nêu $T(n)$, được mở rộng dần trong (**b**)-(**d**) để tạo thành cây đệ quy. Cây được khai triển đầy đủ trong phần (**d**) có chiều cao lg n (nó có lg n + 1 cấp).

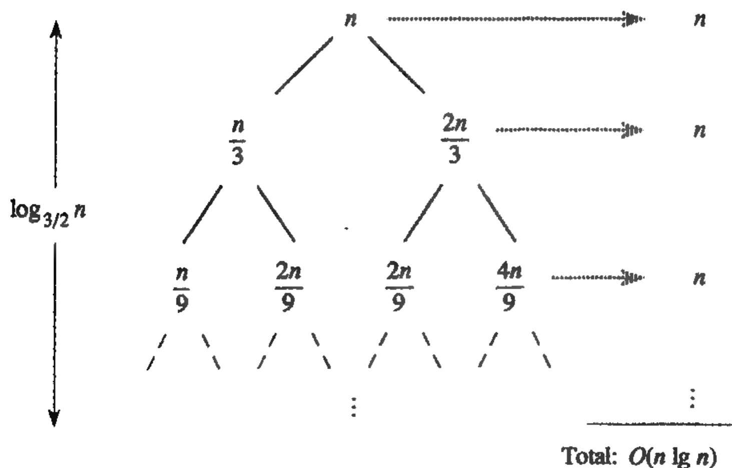
Bài tập

4.2-1

Xác định một cận trên tiệm cận tốt trên phép truy toán $T(n) = 3T(\lfloor n/2 \rfloor + n$ bằng phép lặp lại.

4.2-2

Biện luận nghiệm cho phép truy toán $T(n) = T(n/3) + T(2n/3) + n$ là $\Omega(n \lg n)$ bằng cách sử dụng một cây đệ quy.



Hình 4.2 Một cây đệ quy cho phép truy toán $T(n) = T(n/3) + T(2n/3) + n$.

4.2-3

Vẽ cây đệ quy cho $T(n) = 4T(\lfloor n/2 \rfloor) + n$, và cung cấp các cận sát tiệm cận trên nghiệm của nó.

4.2-4

Dùng phép lặp lại để giải quyết phép truy toán $T(n) = T(n - a) + T(a) + n$, ở đó $a \geq 1$ là một hằng.

4.2-5

Dùng một cây đệ quy để giải quyết phép truy toán $T(n) = T(\alpha n)+T((1-\alpha)n) + n$, ở đó α là một hằng trong miền $0 < \alpha < 1$.

4.3 Phương pháp chủ

Phương pháp chủ cung cấp một phương pháp kiểu "sách nấu ăn" để giải các phép truy toán có dạng

$$T(n) = aT(n/b) + f(n), \qquad (4.5)$$

ở đó $a \geq 1$ và $b > 1$ là các hằng và $f(n)$ là một hàm dương theo tiệm cận. Phương pháp chủ yêu cầu phải nhớ ba trường hợp, nhưng sau đó nghiệm của nhiều phép truy toán có thể được xác định khá dễ dàng, thường không cần đến giấy bút.

Phép truy toán (4.5) mô tả thời gian thực hiện của một thuật toán chia bài toán có kích cỡ n thành a bài toán con, mỗi bài có kích cỡ n/b, ở đó a và b là các hằng dương. a bài toán con được giải theo đệ quy, mỗi bài trong thời gian $T(n/b)$. Hao phí của việc chia bài toán và tổ hợp các kết quả của các bài toán con được mô tả bằng hàm $f(n)$. (Nghĩa là, dùng hệ ký hiệu trong Đoạn 1.3.2, $f(n) = D(n) + C(n)$.) Ví dụ, phép truy toán nảy sinh từ thủ tục MERGE-SORT có $a = 2$, $b = 2$, và $f(n) = \Theta(n)$.

Xét theo tính đúng đắn kỹ thuật, phép truy toán thực tế không được định nghĩa tốt bởi n/b có thể không là một số nguyên. Tuy nhiên, việc thay từng trong số a số hạng $T(n/b)$ bằng $T(\lfloor n/b \rfloor)$ hoặc $T(\lceil n/b \rceil)$ sẽ không ảnh hưởng đến cách ứng xử tiệm cận của phép truy toán. (Ta sẽ chứng minh điều này trong đoạn sau.) Do đó, để tiện dụng, ta thường bỏ qua các hàm sàn và trần khi viết các phép truy toán chia để trị theo dạng này.

Định lý chủ

Phương pháp chủ tùy thuộc vào định lý sau.

Định lý 4.1 (Định lý chủ)

Cho $a \geq 1$ và $b > 1$ là các hằng, cho $f(n)$ là một hàm, và $T(n)$ được định nghĩa trên các số nguyên không âm theo phép truy toán

$$T(n) = aT(n/b) + f(n),$$

ở đó ta diễn dịch n/b theo nghĩa $\lfloor n/b \rfloor$ hoặc $\lceil n/b \rceil$. Như vậy $T(n)$ có thể được định cận theo tiệm cận như sau.

1. Nếu $f(n) = O(n^{\log_b a - \epsilon})$ với một hằng $\epsilon > 0$ nào đó, thì $T(n) = \Theta(n^{\log_b a})$.

2. Nếu $f(n) = \Theta(n^{\log_b a})$, thì $T(n) = \Theta(n^{\log_b a} \lg n)$.

3. Nếu $f(n) = \Omega(n^{\log_b a + \epsilon})$ với một hằng $\epsilon > 0$ nào đó, và nếu $af(n/b) \leq cf(n)$ với một hằng $c < 1$ nào đó và tất cả n đủ lớn, thì $T(n) = \Theta(f(n))$

Trước khi áp dụng định lý chủ cho vài ví dụ, ta bỏ chút thời gian để tìm hiểu ý nghĩa của nó. Trong cả ba trường hợp, ta đang so sánh hàm $f(n)$ với hàm $n^{\log_b a}$. Theo trực giác, nghiệm cho phép truy toán được xác định bởi hàm lớn hơn trong số hai hàm. Như trong trường hợp 1, nếu hàm $n^{\log_b a}$ lớn hơn, nghiệm sẽ là $T(n) = \Theta(n^{\log_b a})$. Như trong trường hợp 3, nếu hàm $f(n)$ lớn hơn, nghiệm sẽ là $T(n) = \Theta(f(n))$. Như trong trường hợp 2, nếu hai hàm cùng kích cỡ, ta nhân với một thừa số lôga, và nghiệm sẽ là $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(f(n) \lg n)$.

Ngoài trực giác này, có vài chi tiết kỹ thuật cần được hiểu rõ. Trong trường hợp thứ nhất, không những $f(n)$ phải nhỏ hơn $n^{\log_b a}$, nó còn phải nhỏ hơn *theo đa thức*. Nghĩa là, theo tiệm cận $f(n)$ phải nhỏ hơn $n^{\log_b a}$ một thừa số n^{ϵ} với một hằng $\epsilon > 0$ nào đó. Trong trường hợp thứ ba, không những $f(n)$ phải lớn hơn $n^{\log_b a}$, nó còn phải lớn hơn theo đa thức và ngoài ra còn phải thỏa điều kiện "tính đều" ["regularity"] $af(n/b) \leq cf(n)$. Điều kiện này được thỏa bởi hầu hết các hàm được định cận theo đa thức mà ta sẽ gặp.

Điều quan trọng là phải nhận thức rõ ba trường hợp này không bao hàm tất cả mọi khả năng cho $f(n)$. Có một lỗ hổng giữa các trường hợp 1 và 2 khi $f(n)$ nhỏ hơn $n^{\log_b a}$ nhưng không nhỏ hơn theo đa thức. Cũng vậy, có một lỗ hổng giữa các trường hợp 2 và 3 khi $f(n)$ lớn hơn $n^{\log_b a}$ nhưng không lớn hơn theo đa thức. Nếu hàm $f(n)$ rơi vào một trong các lỗ hổng này, hoặc giả điều kiện tính đều trong trường hợp 3 không được duy trì, ta không thể dùng phương pháp chủ để giải quyết phép truy toán.

Dùng phương pháp chủ

Để dùng phương pháp chủ, chỉ việc xác định áp dụng trường hợp nào của định lý chủ (nếu có) và viết ra đáp án.

Để lấy ví dụ đầu tiên, hãy xét

$T(n) = 9T(n/3) + n$.

Với phép truy toán này, ta có $a = 9$, $b = 3$, $f(n) = n$, và như vậy $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Bởi $f(n) = O(n^{\log_3 9-\epsilon})$, ở đó $\epsilon = 1$, nên ta có thể áp dụng trường hợp 1 của định lý chủ và kết luận nghiệm là $T(n) = \Theta(n^2)$.

Giờ đây hãy xét

$T(n) = T(2n/3) + 1$,

ở đó $a = 1$, $b = 3/2$, $f(n) = 1$, và $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Trường hợp 2 áp dụng được, bởi $f(n) = \Theta(n^{\log_3 9-\epsilon}) = \Theta(1)$, và như vậy nghiệm cho phép truy toán là $T(n) = \Theta(\lg n)$.

Với phép truy toán

$T(n) = 3T(n/4) + n\lg n$,

ta có $a = 3$, $b = 4$, $f(n) = n \lg n$, và $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$. Do $f(n) = \Omega(n^{\log_4 3+\epsilon}) = \Theta(1)$ ở đó $\epsilon \approx 0.2$, nên trường hợp 3 áp dụng được nếu ta có thể chứng tỏ điều kiện tính đều áp dụng cho $f(n)$. Với n đủ lớn, $af(n/b) = 3(n/4)\lg(n/4) \leq (3/4)n \lg n = cf(n)$ với $c = 3/4$. Bởi vậy, theo trường hợp 3, nghiệm cho phép truy toán là $T(n) = \Theta(n \lg n)$.

Phương pháp chủ không áp dụng cho phép truy toán

$T(n) = 2T(n/2) + n \lg n$,

cho dù nó có dạng thức đúng: $a = 2$, $b = 2$, $f(n) = n \lg n$, và $n^{\log_b a} = n$. Dường như trường hợp 3 sẽ áp dụng, bởi theo tiệm cận $f(n) = n \lg n$ lớn hơn $n^{\log_b a} = n$ nhưng không lớn hơn *theo đa thức*. Theo tiệm cận, tỷ số $f(n)/n^{\log_b a} = (n\lg n)/n = \lg n$ nhỏ hơn n^ϵ với hằng dương ϵ bất kỳ. Bởi vậy, phép truy toán rơi vào lỗ hổng giữa trường hợp 2 và trường hợp 3. (Xem Bài tập 4.4-2 để có giải pháp.)

Bài tập

4.3-1

Dùng phương pháp chủ để cung cấp các cận sát tiệm cận với các phép truy toán dưới đây.

a. $T(n) = 4T(n/2) + n$.

b. $T(n) = 4T(n/2) + n^2$.

c. $T(n) = 4T(n/2) + n^3$.

4.3-2

Thời gian thực hiện của một thuật toán A được mô tả bởi phép truy toán $T(n) = 7T(n/2) + n^2$. Một thuật toán ganh đua A' có một thời gian thực hiện $T'(n) = aT'(n/4) + n^2$. Đâu là giá trị số nguyên lớn nhất cho a sao cho A' nhanh hơn A theo tiệm cận?

4.3-3

Dùng phương pháp chủ để chứng tỏ nghiệm cho phép truy toán $T(n) = T(n/2) + \Theta(1)$ của kỹ thuật tìm nhị phân (xem Bài tập 1.3-5) là $T(n) = \Theta(\lg n)$.

4.3-4

Xem xét điều kiện tính đều [regularity] $af(n/b) \leq cf(n)$ với một hằng $c < 1$ nào đó, là một phần trường hợp 3 của định lý chủ. Nêu một ví dụ về một hàm $f(n)$ đơn giản thỏa tất cả các điều kiện trong trường hợp 3 của định lý chủ ngoại trừ điều kiện tính đều.

* 4.4 Chứng minh định lý chủ

Đoạn này mô tả một phép chứng minh của định lý chủ (Định lý 4.1) dành cho bạn đọc cao cấp hơn. Không cần phải hiểu rõ phép chứng minh để áp dụng định lý.

Phép chứng minh bao gồm hai phần. Phần đầu phân tích phép truy toán "chủ" (4.5), dưới giả thiết giản lược rằng $T(n)$ chỉ được định nghĩa trên các lũy thừa chính xác của $b > 1$, nghĩa là, với $n = 1, b, b^2,...$. Phần này cung cấp toàn bộ trực giác cần thiết để hiểu rõ tại sao định lý chủ lại đúng. Phần hai nêu cách mở rộng phép phân tích theo tất cả các số nguyên dương n và cách áp dụng kỹ thuật đơn thuần toán học cho bài toán điều quản cận dưới và cận trên.

Trong đoạn này, đôi lúc ta có phần lạm dụng hệ ký hiệu tiệm cận bằng cách dùng nó để mô tả cách ứng xử của các hàm chỉ được định nghĩa trên các lũy thừa chính xác của b. Chắc bạn còn nhớ, các phần định nghĩa của các hệ ký hiệu tiệm cận yêu cầu chứng minh các cận với tất cả các con số đủ lớn, chứ không chỉ với các lũy thừa của b. Bởi ta có thể tạo các hệ ký hiệu tiệm cận mới áp dụng cho tập hợp $\{b^i : i = 0, 1, ...\}$, thay vì các số nguyên không âm, lạm dụng này là không quan trọng.

Tuy vậy, ta phải luôn đề phòng khi đang dùng hệ ký hiệu tiệm cận trên một lĩnh vực giới hạn sao cho ta không rút ra các kết luận sai. Ví dụ, chứng minh rằng $T(n) = O(n)$ khi n là một lũy thừa chính xác của 2 sẽ không bảo đảm rằng $T(n) = O(n)$. Hàm $T(n)$ có thể được định nghĩa là

$$T(n) = \begin{cases} n & \text{nếu } n = 1, 2, 4, 8, \ldots, \\ n^2 & \text{bằng không}, \end{cases}$$

trong trường hợp đó cận trên tốt nhất có thể được chứng minh sẽ là $T(n) = O(n^2)$. Do kiểu hệ quả quyết liệt này, ta sẽ không bao giờ dùng

hệ ký hiệu tiệm cận trên một lĩnh vực giới hạn mà không làm thật rõ ngữ cảnh mà ta đang làm thế.

4.4.1 Phép chứng minh với các lũy thừa chính xác

Phần đầu của phép chứng minh định lý chủ sẽ phân tích phép truy toán chủ (4.5),

$$T(n) = aT(n/b) + f(n),$$

dưới giả thiết n là một lũy thừa chính xác của $b > 1$, ở đó b không nhất thiết phải là một số nguyên. Tiến trình phân tích được tách thành ba bổ đề [lemmas]. Bổ đề thứ nhất rút gọn bài toán giải quyết phép truy toán chủ thành bài toán đánh giá một biểu thức chứa một phép lấy tổng. Bổ đề thứ hai xác định các cận trên phép lấy tổng này. Bổ đề thứ ba gom chung hai bổ đề đầu để chứng minh một phiên bản của định lý chủ cho trường hợp ở đó n là một lũy thừa chính xác của b.

Bổ đề 4.2

Cho $a \geq 1$ và $b > 1$ là các hằng, và $f(n)$ là một hàm không âm được định nghĩa trên các lũy thừa chính xác của b. Định nghĩa $T(n)$ trên các lũy thừa chính xác của b bằng phép truy toán

$$T(n) = \begin{cases} \Theta(1) & \text{nếu } n = 1, \\ aT(n/b) + f(n) & \text{nếu } n = b^i, \end{cases}$$

ở đó i là một số nguyên dương. Vậy

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n-1} a^j f(n/b^j). \tag{4.6}$$

Chứng minh Việc lặp lại phép truy toán cho ra

$$\begin{aligned} T(n) &= f(n) + aT(n/b) \\ &= f(n) + af(n/b) + a^2T(n/b^2) \\ &= f(n) + af(n/b) + a^2f(n/b^2) + \ldots \\ &\quad + a^{\log_b n-1} f(n/b^{\log_b n-1}) + a^{\log_b n}T(1). \end{aligned}$$

Bởi $a^{\log_b n} = n^{\log_b a}$, nên số hạng cuối của biểu thức này trở thành

$$a^{\log_b n}T(1) = \Theta(n^{\log_b a}),$$

dùng điều kiện biên $T(1) = \Theta(1)$. Các số hạng còn lại có thể được diễn tả dưới dạng tổng

$$\sum_{j=0}^{\log_b n-1} a^j f(n/b^j);$$

như vậy,

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n-1} a^j f(n/b^j) ,$$

hoàn tất phép chứng minh.

Cây đệ quy

Trước khi tiếp tục, ta thử dùng cây đệ quy để phát triển vài trực giác. Hình 4.3 nêu cây tương ứng với lần lặp lại của phép truy toán trong Bổ đề 4.2. Gốc của cây có hao phí $f(n)$, và nó có a cây con [children], mỗi con có hao phí $f(n/b)$. (Để tiện dụng, ta xem a như là một số nguyên, nhất là khi đang hình dung cây đệ quy, song toán học không yêu cầu điều đó.) Mỗi con có a con có hao phí $f(n/b^2)$, và như vậy có a^2 nút [nodes] cách gốc khoảng cách 2. Nói chung, có a^j nút cách gốc khoảng cách j, và mỗi nút có hao phí $f(n/b^j)$. Hao phí của mỗi lá là $T(1) = \Theta(1)$, và mỗi lá cách gốc khoảng cách $\log_b n$, bởi $n/b^{\log_b n} = 1$. Có $a^{\log_b a} = n^{\log_a b}$ lá trong cây.

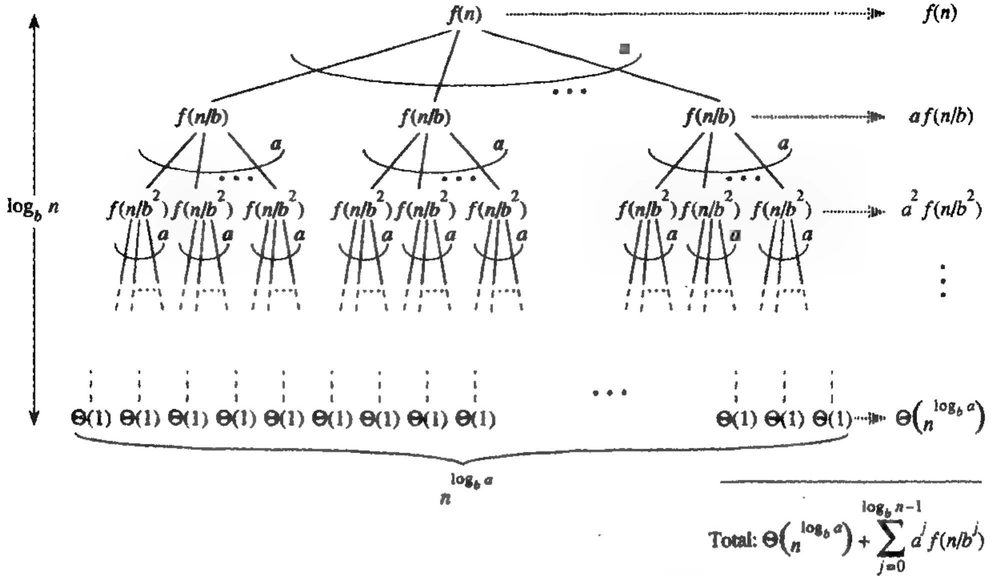
Ta có thể có phương trình (4.6) bằng cách lấy tổng các hao phí của mỗi cấp của cây, xem hình minh họa. Hao phí của một cấp j gồm các nút nội tại là $a^j f(n/b^j)$, và do đó tổng tất cả các cấp nút nội tại là

$$\sum_{j=0}^{\log_b n-1} a^j f(n/b^j) .$$

Trong thuật toán chia để trị cơ sở, tổng này biểu thị cho các hao phí của tiến trình chia các bài toán thành các bài toán con rồi tái hợp các bài toán con. Hao phí của tất cả các lá, là hao phí để thực hiện tất cả $n\log_b^a$ bài toán con có kích cỡ 1, là $\Theta(n^{\log_b a})$.

Theo dạng cây đệ quy, ba trường hợp của định lý chủ tương ứng với các trường hợp ở đó tổng hao phí của cây (1) bị chi phối bởi các hao phí trong các lá, (2) được phân phối đều qua các cấp của cây, hoặc (3) bị chi phối bởi hao phí của gốc.

Phép lấy tổng trong phương trình (4.6) mô tả hao phí của tiến trình chia và tổ hợp các bước trong thuật toán chia để trị cơ sở. Bổ đề kế tiếp cung cấp các cận tiệm cận trên sự tăng trưởng của phép lấy tổng.



Hình 4.3 Cây đệ quy được phát sinh bởi $T(n) = aT(n/b) + f(n)$. Cây là một cây thuộc a [a-ary] hoàn chỉnh có $n^{\log_b a}$ lá và chiều cao $\log_b a$. Hao phí của mỗi cấp được nêu bên phải, và tổng của chúng được nêu trong phương trình (4.6).

***Bổ đề* 4.3**

Cho $a \geq 1$ và $b > 1$ là các hằng, và $f(n)$ là một hàm không âm được định nghĩa trên các lũy thừa chính xác của b. Một hàm $g(n)$ được định nghĩa trên các lũy thừa chính xác của b bởi

$$g(n) = \sum_{j=0}^{\log_b n-1} a^j f(n/b^j). \tag{4.7}$$

sau đó có thể được định cận theo tiệm cận với các lũy thừa chính xác của b như sau.

1. Nếu $f(n) = O(n^{\log_b a-\epsilon})$ với một hằng $\epsilon > 0$ nào đó, thì $g(n) = O(n^{\log_b a})$.

2. Nếu $f(n) = \Theta(n^{\log_b a})$, thì $g(n) = \Theta(n^{\log_b a} \lg n)$.

3. Nếu $af(n/b) \leq cf(n)$ với một hằng $c < 1$ nào đó và tất cả $n \geq b$, thì $g(n) = \Theta(f(n))$.

Chứng minh Với trường hợp 1, ta có $f(n) = O(n^{\log_b a-\epsilon})$, hàm ý $f(n/b^j) = O((n/b^j)^{\log_b a-\epsilon})$. Thay vào phương trình (4.7) cho ra

$$g(n) = O\left(\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a-\epsilon}\right). \tag{4.8}$$

Ta định cận phép lấy tổng trong hệ ký hiệu O bằng cách đưa ra ngoài ngoặc các số hạng và rút gọn, để lại một chuỗi cấp số nhân tăng:

$$\begin{aligned}\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a-\epsilon} &= n^{\log_b a-\epsilon}\sum_{j=0}^{\log_b n-1}\left(\frac{ab^\epsilon}{b^{\log_b a-\epsilon}}\right)^j \\ &= n^{\log_b a-\epsilon}\sum_{j=0}^{\log_b n-1}(b^\epsilon)^j \\ &= n^{\log_b a-\epsilon}\left(\frac{b^{\epsilon\log_b n}-1}{b^\epsilon-1}\right) \\ &= n^{\log_b a-\epsilon}\left(\frac{n^\epsilon-1}{b^\epsilon-1}\right).\end{aligned}$$

Do b và ϵ là các hằng, nên biểu thức cuối rút gọn thành $n^{\log_b a-\epsilon} O(n^\epsilon) = O(n^{\log_b a})$. Thay biểu thức này cho phép lấy tổng trong phương trình (4.8) cho ra

$$g(n) = O(n^{\log_b a}),$$

và trường hợp 1 được chứng minh.

Dưới giả thiết $f(n) = \Theta(n^{\log_b a})$ cho trường hợp 2, ta có $f(n/b^j) = \Theta((n/b^j)^{\log_b a})$. Thay vào phương trình (4.7) cho ra

$$g(n) = \Theta\left(\sum_{j=0}^{\log_b n-1} a^j\left(\frac{n}{b^j}\right)^{\log_b a}\right) .$$

Ta định cận phép lấy tổng trong Θ như trong trường hợp 1, nhưng lần này ta không có một chuỗi cấp số nhân. Thay vì thế, ta phát hiện mọi số hạng của phép lấy tổng là như nhau:

$$\begin{aligned}\sum_{j=0}^{\log_b n-1} a^j\left(\frac{n}{b^j}\right)^{\log_b a-\epsilon} &= n^{\log_b a-\epsilon}\sum_{j=0}^{\log_b n-1}\left(\frac{a}{b^{\log_b a}}\right)^j \\ &= n^{\log_b a-\epsilon}\sum_{j=0}^{\log_b n-1} 1 \\ &= n^{\log_b a-\epsilon}\log_b n .\end{aligned}$$

Thay biểu thức này cho phép lấy tổng trong phương trình (4.9) cho ra

$$\begin{aligned}g(n) &= \Theta(n^{\log_b a-\epsilon}\log_b n) \\ &= \Theta(n^{\log_b a-\epsilon}\log n) ,\end{aligned}$$

và trường hợp 2 được chứng minh.

Trường hợp 3 cũng được chứng minh như vậy. Do $f(n)$ xuất hiện trong phần định nghĩa (4.7) của $g(n)$ và tất cả các số hạng của $g(n)$ không âm, nên ta có thể kết luận $g(n) = \Omega(f(n))$ với các lũy thừa chính xác của b. Dưới giả thiết $af(n/b) \leq cf(n)$ với một hằng $c < 1$ nào đó và tất cả $n \geq b$, ta có $a^j f(n/b^j) \leq c^j f(n)$.

Thay vào phương trình (4.7) và rút gọn sẽ cho ra một chuỗi cấp số

nhân, nhưng khác với các chuỗi trong trường hợp 1, chuỗi có các số hạng giảm:

$$\begin{aligned}
g(n) &\leq \sum_{j=0}^{\log_b n-1} a^j f(n/b^j) \\
&\leq \sum_{j=0}^{\log_b n-1} c^j f(n) \\
&\leq f(n) \sum_{j=0}^{\infty} c^j \\
&= f(n)\left(\frac{1}{1-c}\right) \\
&= O(f(n)),
\end{aligned}$$

bởi c là hằng. Như vậy, ta có thể kết luận $g(n) = \Theta(f(n))$ với các lũy thừa chính xác của b. Trường hợp 3 được chứng minh, hoàn thành phép chứng minh của bổ đề.

Giờ đây ta có thể chứng minh một phiên bản của định lý chủ cho trường hợp ở đó n là một lũy thừa chính xác của b.

Bổ đề 4.4

Cho $a \leq 1$ và $b > 1$ là các hằng, và $f(n)$ là một hàm không âm được định nghĩa trên các lũy thừa chính xác của b. Định nghĩa $T(n)$ trên các lũy thừa chính xác của b bằng phép truy toán

$$T(n) = \begin{cases} \Theta(1) & \text{nếu } n = 1\ , \\ aT(n/b) + f(n) & \text{nếu } n = b^i, \end{cases}$$

ở đó i là một số nguyên dương. Vậy $T(n)$ có thể được định cận theo tiệm cận với các lũy thừa chính xác của b như sau.

1. Nếu $f(n) = O(n^{\log_b a-\epsilon})$ với một hằng $\epsilon > 0$ nào đó, thì $T(n) = \Theta(n^{\log_b a})$.

2. Nếu $f(n) = \Theta(n^{\log_b a})$, thì $T(n) = \Theta(n^{\log_b a}\lg n)$.

3. Nếu $f(n) = \Omega(n^{\log_b a+\epsilon})$ với một hằng $\epsilon > 0$ nào đó, và nếu $af(n/b) \leq cf(n)$ với một hằng $c < 1$ nào đó và tất cả n đủ lớn, thì $T(n) = \Theta(f(n))$.

Chứng minh Ta dùng các cận trong Bổ đề 4.3 để đánh giá phép lấy tổng (4.6) từ Bổ đề 4.2. Với trường hợp 1, ta có

$$\begin{aligned}
T(n) &= \Theta(n^{\log_b a}) + \Theta(f(n)) \\
&= \Theta(f(n)).
\end{aligned}$$

và với trường hợp 2,

$$\begin{aligned}
T(n) &= \Theta(n^{\log_b a}) + \Theta(n^{\log_b a}\lg n) \\
&= \Theta(n^{\log_b a}\lg n)\ .
\end{aligned}$$

Với trường hợp 3, điều kiện $af(n/b) \le cf(n)$ hàm ý $f(n) = \Omega 2(n^{\log_b a+\epsilon})$ (xem Bài tập 4.4-3). Bởi vậy,

$$T(n) = \Theta(n^{\log_b a}) + \Theta(n^{\log_b a}) = \Theta(n^{\log_b a}),$$

4.4.2 Cận dưới và cận trên

Để hoàn tất phép chứng minh định lý chủ, giờ đây ta phải mở rộng tiến trình phân tích theo tình huống ở đó cận dưới và cận trên được dùng trong phép truy toán chủ, sao cho phép truy toán được định nghĩa cho tất cả các số nguyên, chứ không chỉ các lũy thừa chính xác của b. Việc thu được một cận dưới trên

$$T(n) = aT(\lceil n/b \rceil) + f(n) \qquad (4.10)$$

và một cận trên trên

$$T(n) = aT(\lfloor n/b \rfloor) + f(n) \qquad (4.11)$$

là thường lệ, bởi cận $\lceil n/b \rceil \ge n/b$ có thể được đẩy qua trong trường hợp thứ nhất để cho ra kết quả thỏa đáng, và cận $\lfloor n/b \rfloor \le n/b$ có thể được đẩy qua trong trường hợp thứ hai. Việc định cận dưới phép truy toán (4.11) yêu cầu kỹ thuật tương tự như việc định cận trên phép truy toán (4.10), do đó ta chỉ trình bày cận sau mà thôi.

Ta muốn lặp lại phép truy toán (4.10), như trong Bổ đề 4.2. Khi lặp lại phép truy toán, ta thu được một dãy các dẫn chứng đệ quy trên các đối số

$$\begin{aligned} & n, \\ & \lceil n/b \rceil \\ & \lceil \lceil n/b \rceil /b \rceil, \\ & \lceil \lceil \lceil n/b \rceil /b \rceil /b \rceil, \\ & \vdots \end{aligned}$$

Ta hãy thể hiện thành phần thứ i trong dãy bằng n_i, ở đó

$$n_i = \begin{cases} n & \text{nếu } i = 0, \\ \lceil n_{i-1}/b \rceil & \text{nếu } i > 0. \end{cases} \qquad (4.12)$$

Mục tiêu đầu tiên của ta là xác định số lần lặp k sao cho n_k là một hằng. Dùng bất đẳng thức $\lceil x \rceil \le x + 1$, ta có

$$\begin{aligned} n_0 &\le n, \\ n_1 &\le \frac{n}{b} + 1, \\ n_2 &\le \frac{n}{b^2} + \frac{1}{b} + 1, \end{aligned}$$

$$n_3 \le \frac{n}{b^3} + \frac{1}{b^2} + \frac{1}{b} + 1,$$

.
.
.

Nói chung,

$$\begin{aligned} n_i &\le \frac{n}{b^i} + \sum_{j=0}^{i-1} \frac{1}{b_j} \\ &\le \frac{n}{b^i} + \frac{b}{b-1} \end{aligned}$$

và như vậy, khi $i = \lfloor \log_b n \rfloor$, ta có $n_i \le b + b/(b-1) = O(1)$.

Giờ đây ta có thể lặp lại phép truy toán (4.10), thu được

$$\begin{aligned} T(n) &= f(n_0) + aT(n_1) \\ &= f(n_0) + af(n_1) + a^2T(n_2) \\ &\le f(n_0) + af(n_1) + a^2f(n_2) \\ &\quad + a^{\lfloor \log_b n \rfloor - 1} f(n_{\lfloor \log_b n \rfloor - 1}) + a^{\lfloor \log_b n \rfloor} T(n_{\lfloor \log_b n \rfloor}) \\ &= \Theta(n^{\log_b a}) + \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j f(n_j), \end{aligned} \tag{4.13}$$

tương tự như phương trình (4.6), ngoại trừ n là một số nguyên tùy ý và không bị giới hạn là một lũy thừa chính xác của b.

Giờ đây ta có thể đánh giá phép lấy tổng

$$g(n) = \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j f(n_j) \tag{4.14}$$

từ (4.13) theo cách tương tự như phép chứng minh của Bổ đề 4.3. Bắt đầu bằng trường hợp 3, nếu $af(\lceil n/b \rceil) \le cf(n)$ với $n > b + b/(b-1)$, ở đó $c < 1$ là một hằng, thì nó theo $a^j f(n_j) \le c^j f(n)$. Như vậy, tổng trong phương trình (4.14) có thể được đánh giá hệt như trong Bổ đề 4.3. Với trường hợp 2, ta có $f(n) = \Theta(n^{\log_b a})$. Nếu có thể chứng tỏ $f(n_j) = O(n^{\log_b a}/a^j) = O((n/b^j)^{\log_b a})$, thì phép chứng minh cho trường hợp 2 của Bổ đề 4.3 sẽ trót lọt. Nhận thấy $j \le \lceil \log_b n \rceil$ hàm ý $b^j/n \le 1$. Cận $f(n) = O(n^{\log_b a})$ hàm ý ở đó tồn tại một hằng $c > 0$ sao cho với n_j đủ lớn,

$$\begin{aligned} g(n_j) &\le c\left(\frac{n}{b^j} + \frac{b}{b-1}\right)^{\log_b a} \\ &= c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1 + \left(\frac{b^j}{n} \cdot \frac{b}{b-1}\right)\right)^{\log_b a} \\ &\le c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1 + \frac{b}{b-1}\right)^{\log_b a} \end{aligned}$$

$$\leq O\left(\frac{n^{\log_b a}}{a^j}\right),$$

bởi $c(1 + b/(b - 1))^{\log_b a}$ là một hằng. Như vậy, trường hợp 2 được chứng minh. Phép chứng minh của trường hợp 1 hầu như giống hệt. Điểm chính đó là chứng minh cận $f(n_j) = O(n^{\log_b a-\epsilon})$, tương tự như phép chứng minh tương ứng của trường hợp 2, mặc dù đại số học phức tạp hơn.

Giờ đây ta đã chứng minh các cận trêns trong định lý chủ với tất cả mọi số nguyên n. Phép chứng minh của các cận dưới cũng tương tự.

Bài tập

4.4-1 *

Nêu một biểu thức đơn giản và chính xác với n_i trong phương trình (4.12) cho trường hợp ở đó b là một số nguyên dương thay vì một số thực tùy ý.

4.4-2 *

Chứng tỏ nếu $f(n) = \Theta(n^{\log_b a}\lg^k n)$, ở đó $k \geq 0$, thì phép truy toán chủ có nghiệm $T(n) = \Theta(n^{\log_b a+\epsilon}\lg^{k+1} n)$. Để đơn giản, bạn hạn chế tiến trình phân tích theo các lũy thừa chính xác của b.

4.4-3 *

Chứng tỏ trường hợp 3 của định lý chủ là cường điệu, theo nghĩa điều kiện tính đều $a f(n/b) \leq c f(n)$ với một hằng $c < 1$ nào đó hàm ý ở đó tồn tại một hằng $\epsilon > 0$ sao cho $f(n) = \Omega(n^{\log_b a+\epsilon})$.

Các Bài toán

4-1 Các ví dụ về phép truy toán

Nêu các cận trên và dưới tiệm cận với $T(n)$ trong từng phép truy toán dưới đây. Giả sử $T(n)$ không đổi với $n \leq 2$. Tạo các cận càng sát [tight] càng tốt, và xác minh các đáp án.

a. $T(n) = 2T(n/2) + n^3$.

b. $T(n) = T(9n/10) + n$.

c. $T(n) = 16T(n/4) + n^2$.

d. $T(n) = 7T(n/3) + n^2$.

e. $T(n) = 7T(n/2) + n^2$.

f. $T(n) = 2T(n/4) + \sqrt{n}$.

g. $T(n) = T(n - 1) + n$.

h. $T(n) = T(\sqrt{n}) + 1$.

4-2 Tìm số nguyên thiếu

Một mảng $A[1..n]$ chứa tất cả các số nguyên từ 0 đến n ngoại trừ một. Có thể dễ dàng xác định số nguyên thiếu trong thời gian $O(n)$ bằng cách dùng một mảng phụ trợ $B[0..n]$ để ghi nhận các con số xuất hiện trong A. Tuy nhiên, trong bài toán này, ta không thể truy cập một số nguyên hoàn toàn trong A bằng chỉ một phép toán. Các thành phần của A được biểu thị bằng nhị phân, và phép toán duy nhất mà ta có thể dùng để truy cập chúng là "truy nạp" [fetch] bit thứ j của $A[i]$," bỏ ra một thời gian bất biến.

Chứng tỏ nếu chỉ dùng phép toán này, ta vẫn có thể xác định số nguyên thiếu trong thời gian $O(n)$.

4-3 Các hao phí chuyền tham số

Suốt cuốn sách này, ta mặc nhận tiến trình chuyền tham số trong các lệnh gọi thủ tục bỏ ra một thời gian bất biến, cho dù đang chuyền mảng N-thành phần. Giả thiết này hợp lệ trong hầu hết các hệ thống bởi vì một biến trỏ [pointer] đến mảng được chuyền, chứ không phải chính bản thân mảng. Bài toán này xem xét các kiểu thực thi của ba chiến lược chuyền tham số:

1. Một mảng được chuyền bằng biến trỏ. Thời gian = $\Theta(1)$.

2. Một mảng được chuyền bằng cách chép. Thời gian = $\Theta(N)$, ở đó N là kích cỡ của mảng.

3. Một mảng được chuyền bằng cách chỉ chép miền con [subrange] mà thủ tục được gọi có thể truy cập. Thời gian = $\Theta(p - q + 1)$ nếu mảng con $A[p..q]$ được chuyền.

a. Xem xét the thuật toán tìm nhị phân đệ quy để tìm một con số trong một mảng đã sắp xếp (xem Bài tập 1.3-5). Nêu các phép truy toán với các thời gian thực hiện ca xấu nhất của phương pháp tìm nhị phân khi các mảng được chuyền bằng một trong ba phương pháp trên đây, và nêu các cận trên tốt trên các nghiệm của các phép truy toán. Cho N là kích cỡ của bài toán ban đầu và n là kích cỡ của một bài toán con.

b. Làm lại phần (a) với thuật toán MERGE-SORT trong Đoạn 1.3.1.

4-4 Thêm các ví dụ về phép truy toán

Nêu các cận trên và dưới tiệm cận với $T(n)$ trong từng phép truy toán dưới đây. Giảsử $T(n)$ bất biến với $n \leq 2$. Tạo các cận càng sát càng tốt, và xác minh các đáp án.

a. $T(n) = 3T(n/2) + n \lg n$.

b. $T(n) = 3T(n/3 + 5) + n/2$.

c. $T(n) = 2T(n/2) + n/\lg n$.

d. $T(n) = T(n - 1) + 1/n$.

e. $T(n) = T(n - 1) + \lg n$.

f. $T(n) = \sqrt{n}\, T(\sqrt{n}) + n$.

4-5 Các điều kiện không có hệ thống

Thường, ta có thể định cận một phép truy toán $T(n)$ tại các lũy thừa chính xác của một hằng tích phân b. Bài toán này cung cấp các điều kiện đủ để ta mở rộng cận theo tất cả số thực $n > 0$.

a. Cho $T(n)$ và $h(n)$ là các hàm tăng đơn điệu, và giả sử rằng $T(n) \le h(n)$ khi n là một lũy thừa chính xác của một hằng $b > 1$. Hơn nữa, giả sử rằng $h(n)$ "tăng trưởng chậm chậm" theo nghĩa $h(n) = O(h(n/b))$. Chứng minh $T(n) = O(h(n))$.

b. Giả sử rằng ta có phép truy toán $T(n) = aT(n/b) + f(n)$, ở đó $a \ge 1$, $b > 1$, và $f(n)$ tăng đơn điệu. Giả sử thêm rằng các điều kiện ban đầu của phép truy toán được cung cấp bởi $T(n) = g(n)$ với $n \le n_0$, ở đó $g(n)$ tăng đơn điệu và $g(n_0) \le aT(n_0/b) + f(n_0)$. Chứng minh rằng $T(n)$ tăng đơn điệu.

c. Giản lược phép chứng minh định lý chủ cho trường hợp ở đó $f(n)$ tăng đơn điệu và tăng trưởng chậm chậm. Dùng Bổ đề 4.4.

4-6 Các số Fibonacci

Bài toán này phát triển các tính chất của các số Fibonacci, được định nghĩa bằng phép truy toán (2.13). Ta sẽ dùng kỹ thuật của các hàm sinh để giải quyết phép truy toán Fibonacci. Định nghĩa ***hàm sinh*** [generating function] (hoặc ***chuỗi lũy thừa hình thức*** [formal power series]) $\mathcal{F}$ là

$$\begin{aligned}\mathcal{F}(z) &= \sum_{i=0}^{\infty} F_i z^i \\ &= 0 + z + z^2 + 3z^3 + 3z^4 + 5z^5 + 8z^6 + 13z^7 + 21z^8 + \dots\ .\end{aligned}$$

a. Chứng tỏ $\mathcal{F}(z) = z + z\mathcal{F}(z) + z^2\mathcal{F}(z)$.

b. Chứng tỏ

$$\begin{aligned}\mathcal{F}(z) &= \frac{z}{1 - z - z^2} \\ &= \frac{z}{(1 - \phi z)(1 - \hat{\phi} z)}\end{aligned}$$

$$= \frac{1}{\sqrt{5}}\left(\frac{1}{(1-\phi z)} - \frac{1}{(1-\hat{\phi} z)}\right),$$

ở đó

$$\phi = \frac{1+\sqrt{5}}{2} = 1.61803...$$

và

$$\hat{\phi} = \frac{1-\sqrt{5}}{2} = -0.61803...$$

c. Chứng tỏ

$$\mathcal{F}(z) = \sum_{i=0}^{\infty} \frac{1}{\sqrt{5}}(\phi i - \hat{\phi} i)z^i .$$

d. Chứng minh $F_i = \phi^i / \sqrt{5}$ với $i > 0$, được làm tròn theo số nguyên gần nhất. (*Mách nước* $|\hat{\phi}| < 1$.)

e. Chứng minh $F_{i+2} \geq \phi^i$ for $i \geq 0$.

4-7 Trắc nghiệm chip VLSI

Giáo sư Diogenes có *n* chip VLSI[1] giả định giống nhau mà trên nguyên tắc có thể trắc nghiệm lẫn nhau. Khuôn trắc nghiệm của Giáo sư chỉ có chỗ cho hai chip một lúc. Khi khuôn gá lắp được nạp, chip này trắc nghiệm chip kia và báo cáo nó còn tốt hay bị hỏng. Một chip tốt luôn báo cáo đúng đắn chip kia là tốt hay bị hỏng, nhưng câu trả lời của một chip hỏng không thể tin cậy. Như vậy, bốn kết quả khả dĩ của một trắc nghiệm là như sau:

Chip *A* nói	Chip *B* nói	Kết luận
B là tốt	*A* là tốt	cả hai là tốt, hoặc cả hai là hỏng
B là tốt	*A* là hỏng	ít nhất một bị hỏng
B là hỏng	*A* là tốt	ít nhất một bị hỏng
B là hỏng	*A* là hỏng	ít nhất một bị hỏng

a. Chứng tỏ nếu có trên *n*/2 chip bị hỏng, Giáo sư không thể nhất thiết xác định những chip nào là là tốt bằng bất kỳ chiến lược nào dựa trên kiểu trắc nghiệm theo từng cặp này. Giả sử các chip hỏng có thể hiệp lực để lừa Giáo sư.

b. Xem xét bài toán tìm một chip tốt đơn lẻ trong số *n* chip, giả định trên *n*/2 chip là tốt. Chứng tỏ $\lfloor n/2 \rfloor$ trắc nghiệm theo từng cặp đủ để rút

1 VLSI viết tắt của "very-large-scale integration," là công nghệ chip vi mạch được dùng để chế tạo hầu hết các bộ vi xử lý hiện nay.

gọn bài toán thành một bài có kích cỡ gần phân nửa.

c. Chứng tỏ các chip tốt có thể được định danh bằng $\Theta(n)$ trắc nghiệm theo từng cặp, giả định trên $n/2$ chip là tốt. Nêu và giải phép truy toán mô tả số lần trắc nghiệm.

Ghi chú chương

Các phép truy toán được L. Fibonacci nghiên cứu từ năm 1202; và các số Fibonacci đã được đặt tên theo ông. A. De Moivre đã giới thiệu phương pháp của các hàm sinh (xem Bài toán 4-6) để giải các phép truy toán. Phương pháp chủ được thích ứng từ Bentley, Haken, và Saxe [26], đưa ra phương pháp mở rộng được Bài tập 4.4-2 chứng minh. Knuth [121] và Liu [140] nêu cách giải các phép truy toán tuyến tính bằng phương pháp của các hàm sinh. Purdom và Brown [164] có đề cập mở rộng cách giải phép truy toán.

5 Các Tập Hợp

Trong các chương trước, ta đã mô tả các thành phần của toán rời rạc. Chương này ôn lại đầy đủ hơn các hệ ký hiệu, các phần định nghĩa, và các tính chất căn bản của các tập hợp, các quan hệ, các hàm, các đồ thị, và các cây. Bạn nào đã thành thạo về nội dung này có thể lướt qua.

5.1 Các Tập hợp

Tập hợp là một tập thể các đối tượng có thể phân biệt, có tên ***các phần tử*** hoặc ***các thành phần*** của nó. Nếu một đối tượng x là một phần tử của tập hợp S, ta viết $x \in S$ (đọc là "x là một phần tử của S" hoặc, ngắn gọn hơn, "x nằm trong S"). Nếu x không phải là một phần tử của S, ta viết $x \notin S$. Ta có thể mô tả một tập hợp bằng cách liệt kê rõ ràng các phần tử của nó dưới dạng một danh Sách bên trong các dấu ngoặc ôm. Ví dụ, có thể định nghĩa một tập hợp S chứa chính xác các con số 1, 2, và 3 bằng cách viết $S = \{1, 2, 3\}$. Do 2 là một phần tử của tập hợp S, ta có thể viết $2 \in S$, và bởi 4 không phải là phần tử, nên ta có $4 \notin S$. Một tập hợp không thể chứa cùng đối tượng nhiều lần, và các thành phần của nó không theo thứ tự. Hai tập hợp A và B ***bằng nhau,*** viết là $A = B$, nếu chúng chứa các thành phần giống nhau. Ví dụ, $\{1, 2, 3, 1\} = \{1, 2, 3\} = \{3, 2, 1\}$.

Ta chấp nhận các hệ ký hiệu đặc biệt cho các tập hợp thường gặp.

- $\emptyset$ thể hiện ***tập hợp trống*** [empty set], nghĩa là, tập hợp không chứa phần tử nào.

- **Z** thể hiện tập hợp ***các số nguyên,*** nghĩa là, tập hợp $\{..., -2, -1, 0, 1, 2,...\}$.

- **R** thể hiện tập hợp các ***số thực.***

- **N** thể hiện tập hợp ***các số tự nhiên,*** nghĩa là, tập hợp $\{0, 1, 2,...\}$[1].

Nếu tất cả các thành phần của một tập hợp A được chứa trong tập hợp B, nghĩa là, nếu $x \in A$ hàm ý $x \in B$, ta viết $A \subseteq B$ và nói rằng A là một ***tập hợp con*** của B. Một tập hợp A là một ***tập hợp con riêng*** của B, viết là $A \subset B$, nếu $A \subseteq B$ nhưng $A \neq B$. (Có vài tác giả dùng ký hiệu "$\subset$"

để thể hiện quan hệ tập hợp con thường, thay vì quan hệ tập hợp con riêng.) Với bất kỳ tập hợp A, ta có $A \subseteq A$. Với hai tập hợp A và B, ta có $A = B$ nếu và chỉ nếu $A \subseteq B$ và $B \subseteq A$. Với ba tập hợp bất kỳ A, B, và C, nếu $A \subseteq B$ và $B \subseteq C$, thì $A \subseteq C$. Với tập hợp A bất kỳ, ta có $\emptyset \subseteq A$.

Đôi lúc ta định nghĩa các tập hợp theo nghĩa các tập hợp khác. Cho một tập hợp A, ta có thể định nghĩa một tập hợp $B \subseteq A$ bằng cách phát biểu một tính chất phân biệt các thành phần của B. Ví dụ, có thể định nghĩa tập hợp các số nguyên chẵn là $\{x : x \in \mathbf{Z}$ và $x/2$ là một số nguyên$\}$. Dấu hai chấm trong hệ ký hiệu này được đọc là "sao cho." (Vài tác giả dùng một vạch dọc thay vì dấu hai chấm.)

Cho hai tập hợp A và B, ta cũng có thể định nghĩa các tập hợp mới bằng cách áp dụng các ***phép toán tập hợp*** [set operations]:

- ***Phép giao*** [intersection] của các tập hợp A và B là tập hợp

$A \cap B = \{x : x \in A$ và $x \in B\}$.

- ***Phép hợp*** [union] các tập hợp A và B là tập hợp

$A \cup B = \{x : x \in A$ hoặc $x \in B\}$.

- ***Phép hiệu*** [difference] giữa hai tập hợp A và B là tập hợp

$A - B = \{x : x \in A$ và $x \notin B\}$.

Các phép toán tập hợp tuân thủ các định luật sau đây.

Các định luật tập hợp trống [empty set laws]:

$A \cap \emptyset = \emptyset$,

$A \cup 0 = A$.

Các luật lũy đẳng [idempotency laws]:

$A \cap A = A$,

$A \cup A = A$.

Các luật giao hoán [commutative laws]:

$A \cap B = B \cap A$,

$A \cup B = B \cup A$.

Các luật kết hợp [associative laws]:

$A \cap (B \cap C) = (A \cap B) \cap C$,

$A \cup (B \cup C) = (A \cup B) \cup C$.

Các luật phân phối [distributive laws]:

---

1 Vài tác giả bắt đầu các số tự nhiên bằng 1 thay vì 0. Xu hướng hiện đại dường như bắt đầu bằng 0.

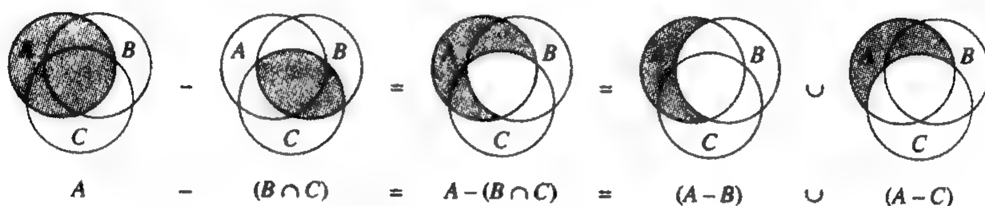
$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C), \quad (5.1)$$
$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C).$$

Các định luật hấp thụ [absorption laws]:

$$A \cap (A \cup B) = A,$$
$$A \cup (A \cap B) = A.$$

Các định luật DeMorgan [DeMorgan's laws]:

$$A - (B \cap C) = (A - B) \cup (A - C), \quad (5.2)$$
$$A - (B \cup C) = (A - B) \cap (A - C).$$



Hình 5.1 Sơ đồ Venn minh họa định luật DeMorgan đầu tiên (5.2). Mỗi tập hợp *A*, *B*, và *C* được biểu thị dưới dạng một vòng tròn trong mặt phẳng.

Định luật DeMorgan đầu tiên, xem Hình 5.1, sử dụng sơ đồ Venn, một hình ảnh đồ họa ở đó các tập hợp được biểu thị dưới dạng các vùng của mặt phẳng.

Thông thường, tất cả các tập hợp đang xem xét đều là những tập hợp con của một tập hợp *U* lớn hơn nào đó có tên ***universe***. Ví dụ, nếu ta đang xem xét các tập hợp khác nhau được tạo thành bằng chỉ các số nguyên, tập hợp **Z** của các số nguyên là một universe thích hợp. Cho một universe *U*, ta định nghĩa ***phần bù*** [complement] của một tập hợp *A* dưới dạng $\bar{A} = U - A$. Với tập hợp bất kỳ $A \subseteq U$, ta có các định luật sau:

$$\bar{\bar{A}} = A,$$
$$A \cap \bar{A} = \emptyset,$$
$$A \cup \bar{A} = U.$$

Các định luật DeMorgan (5.2) có thể được viết lại theo các phần bù. Với hai tập hợp $A, B \subseteq U$ bất kỳ, ta có

$$\overline{A \cap B} = \bar{A} \cup \bar{B},$$
$$\overline{A \cup B} = \bar{A} \cap \bar{B}.$$

Hai tập hợp *A* và *B* ***rời nhau*** nếu chúng không có các thành phần chung, nghĩa là, nếu $A \cup B = \emptyset$. Một tập thể $S = \{S_i\}$ các tập hợp không trống [nonempty] sẽ hình thành một ***phân hoạch*** [partition] của một tập

hợp S nếu các tập hợp ***rời nhau theo từng cặp***, nghĩa là, $S_i, S_j \in S$ và $i \neq j$ hàm ý $S^i \cap S^j = \emptyset$.

và

hợp của chúng là S, nghĩa là,

$$S = \bigcup_{S_i \in S} S_i.$$

Nói cách khác, S hình thành một phân hoạch của S nếu mỗi thành phần của S xuất hiện trong chính xác một $S_i \in S$.

Số lượng các thành phần trong tập hợp được gọi là ***bản số*** [cardinality] (hoặc ***kích cỡ*** [size]) của tập hợp, được ký hiệu là $|S|$. Hai tập hợp có cùng bản số nếu có thể đặt các thành phần của chúng trong mối tương ứng một-một. Bản số của tập hợp trống là $|\emptyset| = 0$. Nếu bản số của một tập hợp là một số tự nhiên, ta nói tập hợp là hữu hạn; bằng không, nó là vô hạn. Một tập hợp vô hạn có thể được đặt trong mối tương ứng một-một với các số tự nhiên **N** ***có thể đếm vô hạn*** [countably infinite]; bằng không, nó ***không đếm được*** [uncountable]. Các số nguyên **Z** là đếm được, nhưng các số thực **R** là không đếm được.

Với hai tập hợp hữu hạn A và B bất kỳ, ta có đồng nhất thức

$$|A \cup B| = |A| + |B| - |A \cap B|, \quad (5.3)$$

từ đó có thể kết luận rằng

$$|A \cup B| \leq |A| + |B|.$$

Nếu A và B rời nhau, thì $|A \cap B| = 0$ và như vậy $|A \cup B| = |A| + |B|$. Nếu $A \subseteq B$, thì $|A| \leq |B|$.

Một tập hợp hữu hạn n thành phần đôi lúc còn được gọi là tập hợp n. Một tập hợp 1 được gọi là một ***tập đơn*** [singleton]. Một tập hợp con k thành phần của một tập hợp đôi lúc được gọi là ***tập hợp con k*** [k-subset].

Tập hợp tất cả các tập con của tập hợp S, kể cả tập hợp trống và chính tập hợp S, được ký hiệu là 2^S và được gọi là ***tập hợp lũy thừa*** [power set] của S. Ví dụ, $2^{\{a,b\}} = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$. Tập hợp lũy thừa của của tập hợp hữu hạn S có bản số $2^{|S|}$

Đôi lúc ta quan tâm đến các cấu trúc giống tập hợp ở đó các thành phần được sắp xếp thứ tự. Một ***cặp có thứ tự*** hai thành phần a và b được ký hiệu là (a, b) và có thể được định nghĩa hình thức là tập hợp $(a, b) = \{a, \{a, b\}\}$. Như vậy, cặp có thứ tự (a, b) không giống với cặp có thứ tự (b, a).

Tích Đề các của hai tập hợp A và B, được ký hiệu là $A \times B$, là tập hợp

của tất cả các cặp có thứ tự sao cho thành phần thứ nhất của cặp là một thành phần của A và thành phần thứ hai là một thành phần của B. Hình thức hơn, $A \times B = \{(a, b): a \in A \text{ and } b \in B\}$.

Ví dụ, $\{a, b\} \times \{a, b, c\} = \{(a, a), (a, b), (a, c), (b, a), (b, b), (b, c)\}$.

Khi A và B là các tập hợp hữu hạn, bản số tích Đề các của chúng là

$$|A \times B| = |A| \cdot |B| . \tag{5.4}$$

Tích Đề các của n tập hợp $A_1, A_2, \ldots, A_n$, là tập hợp của các ***bộ-n*** [n-tuples]

$$A_1 \times A_2 \times \ldots \times A_n = \{(a_1, a_2, \ldots, a_n): a_i \in A_i, i = 1, 2, \ldots, n\} ,$$

bản số của nó là

$$|A_1 \times A_2 \times \ldots \times A_n| = |A_1| \cdot |A_2| \ldots |A_n|$$

nếu tất cả các tập hợp là hữu hạn. Ta thể hiện một tích Đề các gấp-n [n-fold] trên chỉ một tập hợp A bằng tập hợp

$$A^n = A \times A \times \ldots \times A,$$

bản số của nó là $|A^n| = |A|^n$ nếu A là hữu hạn. Một bộ-n cũng có thể được xem là một dãy hữu hạn có chiều dài n (xem trang 97).

Bài tập

5.1-1

Vẽ các sơ đồ Venn minh họa định luật phân phối đầu tiên (5.1).

5.1-2

Chứng minh sự suy rộng của các định luật DeMorgan với một tập thể hữu hạn các tập hợp bất kỳ:

$$\overline{A_1 \cap A_2 \cap \ldots \cap A_n} = \overline{A_1} \cup \overline{A_2} \cup \overline{A_n},$$

$$\overline{A_1 \cup A_2 \cup \ldots \cup A_n} = \overline{A_1} \cap \overline{A_2} \cap \ldots \cap \overline{A_n}.$$

5.1-3 *

Chứng minh sự suy rộng của phương trình (5.3), có tên ***nguyên lý bao hàm và loại trừ*** [principle of inclusion and exclusion]:

$$|A_1 \cup A_2 \cup \ldots \cup A_n| =$$

$$|A_1| + |A_2| + \ldots + |A_n|$$

$$- |A_1 \cap A_2| - |A_1 \cap A_3| - \ldots \text{ (tất cả cặp)}$$

$$+ |A_1 \cap A_2 \cap A_3| + \ldots \text{ (tất cả bộ ba)}$$

$$\vdots$$

$$+ (-1)^{n-1} |A_1 \cap A_2 \cap \ldots \cap A_n| .$$

5.1-4

Chứng tỏ tập hợp các số tự nhiên lẻ là đếm được.

5.1-5

Chứng tỏ với bất kỳ tập hợp hữu hạn S, tập hợp lũy thừa 2^S có $2^{|S|}$ thành phần (nghĩa là, có $2^{|S|}$ tập hợp con riêng biệt của S).

5.1-6

Nêu một định nghĩa quy nạp với một bộ-n [n-tuple] bằng cách mở rộng định nghĩa lý thuyết tập hợp với một cặp có thứ tự.

5.2 Các quan hệ

Một ***quan hệ nhị phân*** [binary relation] R trên hai tập hợp A và B là một tập hợp con của tích Đề các $A \times B$. Nếu $(a, b) \in R$, đôi lúc ta viết $a \, R \, b$. Khi nói R là một quan hệ nhị phân trên một tập hợp A, ta ám chỉ rằng R là một tập hợp con của $A \times A$. Ví dụ, quan hệ "nhỏ hơn" trên các số tự nhiên là tập hợp $\{(a, b) : a, b \in \mathbf{N}$ và $a < b\}$. Một quan hệ thuộc-n [n-ary] trên các tập hợp $A_1, A_2, ..., A_n$ là một tập hợp con của $A_1 \times A_2 \times ... \times A_n$.

Một quan hệ nhị phân $R \subseteq A \times A$ là ***phản xạ*** [reflexive] nếu

$a \, R \, a$

với tất cả $a \in A$. Ví dụ, "=" và "≤" là các quan hệ phản xạ trên **N**, nhưng "<" thì không. Quan hệ R là ***đối xứng*** [symmetric] nếu

$a \, R \, b$ hàm ý $b \, R \, a$

với tất cả $a, b \in A$. Ví dụ, "=" là đối xứng, nhưng "<" và "≤" thì không. Quan hệ R là ***bắc cầu*** [transitive] nếu

$a \, R \, b$ và $b \, R \, c$ hàm ý $a \, R \, c$

với tất cả $a, b, c \in A$. Ví dụ, các quan hệ "<," "≤" và "=" là bắc cầu, nhưng quan hệ $R = \{(a, b) : a, b \in \mathbf{N}$ và $a = b - 1\}$ thì không, bởi $3 \, R \, 4$ và $4 \, R \, 5$ không hàm ý $3 \, R \, 5$.

Một quan hệ là phản xạ, đối xứng, và bắc cầu là một ***quan hệ tương đương*** [equivalence relation]. Ví dụ, "=" là một quan hệ tương đương trên các số tự nhiên, nhưng "<" thì không. Nếu R là một quan hệ tương đương trên một tập hợp A, thì với $a \in A$, ***lớp tương đương*** của a là tập hợp $[a] = \{b \in A : a \, R \, b\}$, nghĩa là, tập hợp tất cả các thành phần tương đương với A. Ví dụ, nếu định nghĩa $R = \{(a, b) : a, b \in \mathbf{N}$ và $a + b$ là một số chẵn$\}$, thì R là một quan hệ tương đương, bởi $a + a$ là chẵn (phản xạ), $a + b$ là chẵn hàm ý $b + a$ là chẵn (đối xứng), và $a + b$ là chẵn và $b + c$ là chẵn hàm ý $a + c$ là chẵn (bắc cầu). Lớp tương đương của 4 là $[4] = \{0,$

2, 4, 6,...}, và lớp tương đương của 3 là [3] = {1, 3, 5, 7,...}. Sau đây là một định lý căn bản về các lớp tương đương.

Định lý 5.1 (Một quan hệ tương đương cũng giống như một phân hoạch)

Các lớp tương đương của bất kỳ quan hệ tương đương R trên một tập hợp A sẽ hình thành một phân hoạch của A, và bất kỳ phân hoạch nào của A đều sẽ xác định một quan hệ tương đương trên A mà các tập hợp trong phân hoạch đó là các lớp tương đương.

Chứng minh

Với phần đầu của phép chứng minh, ta phải chứng tỏ các lớp tương đương của R là không trống, các tập hợp rời nhau theo từng cặp có hợp là A. Bởi vì R là phản xạ, nên $a \in [a]$, và do đó các lớp tương đương là không trống; hơn nữa, bởi mọi thành phần $a \in A$ thuộc về lớp tương đương $[a]$, nên hợp của các lớp tương đương là A. còn lại, ta chứng tỏ rằng các lớp tương đương rời nhau theo từng cặp, nghĩa là, nếu hai lớp tương đương $[a]$ và $[b]$ có một thành phần c chung, thì chúng thực tế cùng một tập hợp. Giờ đây $a\,R\,c$ và $b\,R\,c$, mà theo tính đối xứng và tính bắc cầu hàm ý $a\,R\,b$. Như vậy, với bất kỳ thành phần tùy ý $x \in [a]$, ta có $x\,R\,a$ hàm ý $x\,R\,b$, và như vậy $[a] \subseteq [b]$. Cũng vậy, $[b] \subseteq [a]$, và như vậy $[a] = [b]$.

Với phần hai của phép chứng minh, cho $A = \{A_i\}$ là một phân hoạch của A, và định nghĩa $R = \{(a,b) : \text{ở đó tồn tại } i \text{ sao cho } a \in A_i \text{ và } b \in A_i\}$. Ta biện luận rằng R là một quan hệ tương đương trên A. Phép phản chiếu có thể áp dụng, bởi $a \in A_i$ hàm ý $a\,R\,a$. Tính đối xứng có thể áp dụng, bởi vì nếu $a\,R\,b$, thì a và b nằm trong cùng tập hợp A_i, và do đó $b\,R\,a$. Nếu $a\,R\,b$ và $b\,R\,c$, thì cả ba thành phần đều nằm trong cùng tập hợp, và như vậy $a\,R\,c$ và tính bắc cầu có thể áp dụng. Để biết các tập hợp trong phân hoạch là các lớp tương đương của R, hãy quan sát nếu $a \in A_i$, thì $x \in [a]$ hàm ý $x \in A_i$, và $x \in A_i$ hàm ý $x \in [a]$.

Một quan hệ nhị phân R trên một tập hợp A là ***phản xứng*** [antisymmetric] nếu $a\,R\,b$ và $b\,R\,a$ hàm ý $a = b$.

Ví dụ, quan hệ "≤" trên các số tự nhiên là phản xứng, bởi $a \leq b$ và $b \leq a$ hàm ý $a = b$. Một quan hệ là phản xạ, phản xứng, và bắc cầu là một ***thứ tự từng phần*** [partial order] và ta gọi một tập hợp qua đó định nghĩa một thứ tự từng phần là một ***tập hợp có thứ tự từng phần***. Ví dụ, quan hệ "là một hậu duệ của" là một thứ tự từng phần trên tập hợp tất cả mọi người (nếu ta xem các cá nhân như là những hậu duệ riêng của họ).

Trong một tập hợp có thứ tự từng phần A, có thể không có thành phần "cực đại" ["maximum"] đơn lẻ x sao cho $y\,R\,x$ với tất cả $y \in A$.

Thay vì thế, có thể có vài thành phần ***"tối đa khả dĩ"*** [maximal] x sao cho với không $y \in A$ nào là trường hợp $x R y$. Ví dụ, trong một tập hợp các hộp có kích cỡ khác nhau, có thể có vài hộp tối đa khả dĩ *i* không vừa trong bất kỳ hộp nào khác, nhưng cũng không có một hộp "cực đại" đơn lẻ nào nơi bất kỳ hộp nào khác sẽ vừa.

Một thứ tự từng phần R trên một tập hợp A là một ***thứ tự tuyến tính*** [linear order] hoặc ***thứ tự tổng*** [total order] nếu với tất cả $a, b \in A$, ta có $a R b$ hoặc $b R a$, nghĩa là, nếu mọi kiểu bắt cặp các thành phần của A có thể lập quan hệ R. Ví dụ, quan hệ "$\leq$" là một thứ tự tổng trên các số tự nhiên, nhưng quan hệ "là một hậu duệ của" không phải là một thứ tự tổng trên tập hợp tất cả mọi người, bởi có những cá nhân không có nguồn gốc từ người kia.

Bài tập

5.2-1

Chứng minh quan hệ tập con "$\subseteq$" trên tất cả các tập con của $\mathbf{Z}$ là một thứ tự từng phần chứ không phải là một thứ tự tổng.

5.2-2

Chứng tỏ với bất kỳ số nguyên dương n, quan hệ "tương đương modulo n" là một quan hệ tương đương trên các số nguyên. (Ta nói rằng $a \equiv b \pmod n$ nếu ở đó tồn tại một số nguyên q sao cho $a - b = qn$.) Quan hệ này phân hoạch các số nguyên thành các lớp tương đương nào?

5.2-3

Nêu các ví dụ về các quan hệ

a. phản xạ và đối xứng nhưng không bắc cầu,

b. phản xạ và bắc cầu nhưng không đối xứng,

c. đối xứng và bắc cầu nhưng không phản xạ.

5.2-4

Cho S là một tập hợp hữu hạn, và R là một quan hệ tương đương trên $S \times S$. Chứng tỏ nếu R còn là phản xứng [antisymmetric], thì các lớp tương đương của S đối với R là những tập đơn [singleton].

5.2-5

Giáo sư Narcissus cho là nếu một quan hệ R đối xứng và bắc cầu, thì nó cũng phản xạ. Ông đưa ra phép chứng minh dưới đây. Theo đối xứng, $a R b$ hàm ý $b R a$. Do đó, tính bắc cầu hàm ý $a R a$. Giáo sư có đúng không?

5.3 Các Hàm

Cho hai tập hợp A và B, một ***hàm*** f là một quan hệ nhị phân trên $A \times B$ sao cho với tất cả $a \in A$, ở đó tồn tại chính xác một $b \in B$ sao cho $(a, b) \in f$. Tập hợp A được gọi là miền xác định [domain] của f, và tập hợp B được gọi là đồng miền xác định [codomain] của f. Đôi lúc ta viết $f : A \rightarrow B$; và nếu $(a, b) \in f$, ta viết $b = f(a)$, bởi b được xác định duy nhất bởi sự chọn lựa của a.

Theo trực giác, hàm f gán một thành phần của B cho từng thành phần của A. Không một thành phần nào của A được gán hai thành phần khác nhau của B, nhưng cùng thành phần của B có thể được gán cho hai thành phần khác nhau của A. Ví dụ, quan hệ nhị phân

$f = \{(a, b) : a \in \mathrm{N}$ và $b = a \bmod 2\}$

là một hàm $f : \mathrm{N} \rightarrow \{0, 1\}$, bởi với mỗi số tự nhiên a, ta có chính xác một giá trị b trong $\{0, 1\}$ sao cho $b = a \bmod 2$. Với ví dụ này, $0 = f(0)$, $1 = f(1)$, $0 = f(2)$, vân vân. Ngược lại, quan hệ nhị phân

$g = \{(a,b) : a \in \mathrm{N}$ và $a + b$ là chẵn$\}$

không phải là một hàm, bởi cả hai $(1, 3)$ và $(1, 5)$ đều nằm trong g, và như vậy với chọn lựa $a = 1$, ta không có chính xác một b sao cho $(a, b) \in g$.

Cho một hàm $f : A \rightarrow B$, nếu $b = f(a)$, ta nói rằng a là ***đối số*** [argument] của f và b là ***giá trị*** [value] của f tại a. Có thể định nghĩa một hàm bằng cách nói rõ giá trị của nó với mọi thành phần của miền xác định của nó. Ví dụ, có thể định nghĩa $f(n) = 2n$ với $n \in \mathrm{N}$, có nghĩa là $f = \{(n, 2n) : n \in \mathrm{N}\}$. Hai hàm f và g ***bằng nhau*** nếu chúng có cùng miền xác định và đồng miền xác định và nếu, với tất cả a trong miền xác định, $f(a) = g(a)$.

Một ***dãy hữu hạn*** [finite sequence] có chiều dài n là một hàm f có miền xác định là tập hợp $\{0, 1,..., n - 1\}$. Ta thường biểu thị một dãy hữu hạn bằng cách liệt kê các giá trị của nó: $\langle f(0), f(1),..., f(n-1)\rangle$. Một ***dãy vô hạn*** [infinite sequence] là một hàm có miền xác định là tập hợp N các số tự nhiên. Ví dụ, dãy Fibonacci, được định nghĩa bởi (2.13), là dãy vô hạn $\langle 0, 1, 1, 2, 3, 5, 8, 13, 21,...\rangle$.

Khi miền xác định của một hàm f là một tích Đề các, ta thường bỏ qua các dấu ngoặc đơn phụ trội bao quanh đối số của f. Ví dụ, nếu $f : A_1 \times A_2 \times ... \times A_n \rightarrow B$, ta viết $b = f(a_1, a_2,..., a_n)$ thay vì $b = f((a_1, a_2,..., a_n))$. Ta cũng gọi mỗi a_i là một ***đối số*** với hàm f, tuy về mặt kỹ thuật đối số

(đơn) với f là bộ-n $(a_1, a_2,..., a_n)$.

Nếu $f: A \to B$ là một hàm và $b = f(a)$, thì đôi lúc ta nói rằng b là ảnh của a dưới f. Ảnh của một tập hợp $A' \subseteq A$ dưới f được định nghĩa bởi

$f(A') = \{b \in B: b = f(a)$ với một $a \in A'$ nào đó$\}$.

Miền giá trị [range] của f là ảnh của miền xác định của nó, nghĩa là, $f(A)$. Ví dụ, miền giá trị của hàm $f: \mathbf{N} \to \mathbf{N}$ được định nghĩa bởi $f(n) = 2n$ là $f(\mathbf{N}) = \{m : m = 2n$ với một $n \in \mathbf{N}$ nào đó$\}$.

Một hàm là một ***phép toàn ánh*** nếu miền giá trị của nó là đồng miền xác định của nó. Ví dụ, hàm $f(n) = \lfloor n/2 \rfloor$ là một hàm toàn ánh từ **N** đến **N**, bởi mọi thành phần trong **N** xuất hiện dưới dạng giá trị của f với vài đối số. Ngược lại, hàm $f(n) = 2n$ không phải là một hàm toàn ánh từ **N** đến **N**, bởi không có đối số nào với f có thể cho ra 3 làm một giá trị. Tuy nhiên, hàm $f(n) = 2n$ là một hàm toàn ánh từ các số tự nhiên đến các số chẵn. Một phép toàn ánh $f: A \to B$ đôi lúc được mô tả dưới dạng ánh xạ *A **lên trên** B*. Khi ta nói rằng f là lên trên, ta ám chỉ nó là toàn ánh.

Một hàm $f: A \to B$ là một ***phép đơn ánh*** nếu các đối số riêng biệt với f cho ra các giá trị riêng biệt, nghĩa là, nếu $a \neq a'$ hàm ý $f(a) \neq f(a')$. Ví dụ, hàm $f(n) = 2n$ là một hàm đơn ánh từ **N** đến **N**, bởi mỗi số chẵn b là ảnh dưới f của tối đa một thành phần của miền xác định, tức là $b/2$. Hàm $f(n) = \lfloor n/2 \rfloor$ không phải là đơn ánh, bởi vì giá trị 1 được tạo bởi hai đối số: 2 và 3. Đôi lúc phép đơn ánh còn được gọi là một hàm ***một-một*** [one-to-one function].

Một hàm $f: A \to B$ là một ***phép song ánh*** nếu nó là đơn ánh và toàn ánh. Ví dụ, hàm $f(n) = (-1)^n \lceil n/2 \rceil$ là một phép song ánh từ **N** đến **Z**:

$0 \to 0$,

$1 \to -1$,

$2 \to 1$,

$3 \to -2$,

$4 \to 2$,

.
.
.

Hàm là đơn ánh, bởi không có thành phần nào của **Z** là ảnh của nhiều hơn một thành phần của **N**. Nó là toàn ánh, bởi mọi thành phần của **Z** xuất hiện dưới dạng ảnh của vài thành phần của **N**. Vì thế, hàm là song ánh. Một phép song ánh đôi lúc còn được gọi là một ***tương ứng một-một*** [one-to-one correspondence], bởi nó bắt cặp các thành phần

trong miền xác định và đồng miền xác định. Một phép song ánh từ một tập hợp A đến chính nó đôi lúc được gọi là ***phép hoán vị*** [permutation].

Khi hàm f là song ánh, ***nghịch đảo*** [inverse] của nó f^{-1} được định nghĩa là

$f^{-1}(b) = a$ nếu và chỉ nếu $f(a) = b$.

Ví dụ, nghịch đảo của hàm $f(n) = (-1)^n \lceil n/2 \rceil$ là

$$f^{-1}(m) = \begin{cases} 2m & \text{nếu } m \geq 0, \\ -2m-1 & \text{nếu } m < 0. \end{cases}$$

Bài tập

5.3-1

Cho A và B là các tập hợp hữu hạn, và $f : A \rightarrow B$ là một hàm. Chứng tỏ

a. nếu f là đơn ánh, thì $|A| \leq |B|$;

b. nếu f là toàn ánh, thì $|A| \geq |B|$.

5.3-2

Hàm $f(x) = x + 1$ có song ánh không khi miền xác định [domain] và đồng miền xác định [codomain] là N? Nó có song ánh không khi miền xác định và đồng miền xác định là Z?

5.3-3

Nêu một định nghĩa tự nhiên về nghịch đảo của một quan hệ nhị phân sao cho nếu một quan hệ thực tế là một hàm song ánh, nghịch đảo quan hệ của nó là nghịch đảo phiếm hàm [functional inverse] của nó.

5.3-4 *

Nêu một phép song ánh từ **Z** đến **Z** x **Z**.

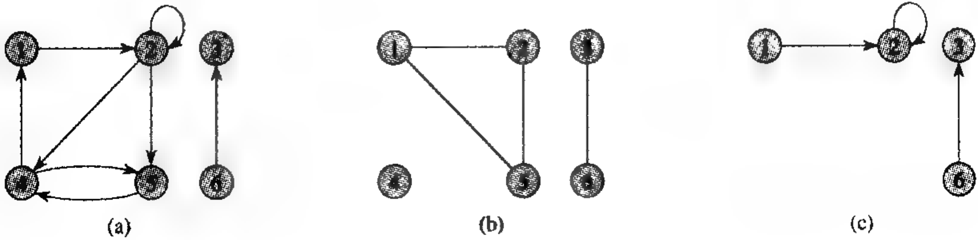
5.4 Đồ thị

Đoạn này trình bày hai kiểu đồ thị: định hướng và không định hướng. bạn đọc sẽ nhận thấy một số định nghĩa trong các tài liệu giáo khoa khác với ở đây, nhưng đa phần, các khác biệt là nhỏ. Đoạn 23.1 nêu cách biểu thị đồ thị trong bộ nhớ máy tính.

Một ***đồ thị có hướng*** (hoặc ***đồ thị định hướng***) G là một cặp (V, E), ở đó V là một tập hợp hữu hạn và E là một quan hệ nhị phân trên V. Tập hợp V được gọi là ***tập hợp đỉnh*** của G, và các thành phần của nó được gọi là ***các đỉnh***. Tập hợp E được gọi là ***tập hợp cạnh*** của G, và các thành phần của nó được gọi là ***các cạnh***. Hình 5.2(a) là một biểu thị hình ảnh

của một đồ thị có hướng trên tập hợp đỉnh {1, 2, 3, 4, 5, 6}. Các đỉnh được biểu thị bởi các vòng tròn trong hình, và các cạnh được biểu thị bởi các mũi tên. Lưu ý, ***các vòng tự lặp*** [self-loops]—các cạnh từ một đỉnh đến chính nó—là có thể.

Trong một ***đồ thị không có hướng*** [undirected graph] $G = (V, E)$, tập hợp cạnh E bao gồm *các* cặp *không có thứ tự* [unordered pairs] của các đỉnh, thay vì các cặp có thứ tự. Nghĩa là, một cạnh là một tập hợp $\{u, v\}$, ở đó $u, v \in V$ và $u \neq v$. Theo quy ước, ta dùng hệ ký hiệu (u, v) cho một cạnh, thay vì hệ ký hiệu tập hợp $\{u, v\}$, và (u, v) và (v, u) được xem là cùng một cạnh. Trong một đồ thị không có hướng, các vòng tự lặp bị cấm, và do đó mọi cạnh bao gồm chính xác hai đỉnh riêng biệt. Hình 5.2(b) là một biểu thị hình ảnh của một đồ thị không có hướng trên tập hợp đỉnh {1, 2, 3, 4, 5, 6}.



Hình 5.2 Đồ thị có hướng và không có hướng. **(a)** Một đồ thị có hướng $G = (V, E)$, ở đó $V = \{1, 2, 3, 4, 5, 6\}$ và $E = \{(1, 2), (2, 2), (2, 4), (2, 5), (4, 1), (4, 5), (5, 4), (6, 3)\}$. Cạnh (2, 2) là một vòng tự lặp. **(b)** Một đồ thị không có hướng $G = (V, E)$, ở đó $V = \{1, 2, 3, 4, 5, 6\}$ và $E = \{(1, 2), (1, 5), (2, 5), (3, 6)\}$. Đỉnh 4 bị cô lập. **(c)** Đồ thị con của đồ thị trong phần (a) được cảm sinh bởi tập hợp đỉnh {1, 2, 3, 6}.

Nhiều định nghĩa không phân biệt đồ thị có hướng và đồ thị không có hướng, mặc dù một số thuật ngữ có ý nghĩa hơi khác trong hai ngữ cảnh. Nếu (u, v) là một cạnh trong một đồ thị có hướng $G = (V, E)$, ta nói rằng (u, v) ***liên thuộc từ*** [incident from] hoặc ***rời*** đỉnh u và ***liên thuộc đến*** [incident to] hoặc ***nhập*** đỉnh v. Ví dụ, các cạnh rời đỉnh 2 trong Hình 5.2(a) là (2, 2), (2, 4), và (2, 5). Các cạnh nhập đỉnh 2 là (1, 2) và (2, 2). Nếu (u, v) là một cạnh trong một đồ thị không có hướng $G = (V, E)$, ta nói rằng (u, v) là ***liên thuộc trên*** [incident on] các đỉnh u và v. Trong Hình 5.2(b), các cạnh liên thuộc trên đỉnh 2 là (1, 2) và (2, 5).

Nếu (u, v) là một cạnh nằm trong một đồ thị $G = (V, E)$, ta nói đỉnh v ***kề với*** đỉnh u. Khi đồ thị không định hướng, quan hệ kề [adjacency relation] là đối xứng. Khi đồ thị được định hướng, quan hệ kề không

nhất thiết đối xứng. Nếu v kề với u trong một đồ thị có hướng, đôi lúc ta viết $u \rightarrow v$. Trong các phần (a) và (b) của Hình 5.2, đỉnh 2 kề với đỉnh 1, bởi cạnh (1, 2) thuộc về cả hai đồ thị. Đỉnh 1 không kề với đỉnh 2 trong Hình 5.2(a), bởi cạnh (2,1) không thuộc về đồ thị.

Độ [degree] của một đỉnh trong một đồ thị không có hướng là số lượng các cạnh liên thuộc trên nó. Ví dụ, đỉnh 2 trong Hình 5.2(b) có độ 2. Trong một đồ thị có hướng, ***độ-ra*** [out-degree] của một đỉnh là số lượng các cạnh rời khỏi nó, và ***độ-vào*** [in-degree] của một đỉnh là số lượng các cạnh nhập vào nó. ***Độ*** của một đỉnh trong một đồ thị có hướng là độ-vào cộng với độ-ra của nó. Đỉnh 2 trong Hình 5.2(a) có độ-vào 2, độ-ra 3, và độ 5.

Một ***lộ trình*** [path] có ***chiều dài k*** từ một đỉnh u đến một đỉnh u' trong một đồ thị $G = (V, E)$ là một dãy $<v_0, v_1, v_2, ..., v_k>$ các đỉnh sao cho $u = v_0$, $u' = v_k$, và $(v_{i-1}, v_i) \in E$ với $i = 1, 2, ..., k$. Chiều dài của lộ trình là số lượng các cạnh trong lộ trình. Lộ trình ***chứa*** các đỉnh $v_0, v_1, ..., v_k$ và các cạnh $(v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k)$. Nếu có một lộ trình p từ u đến u', ta nói rằng u' là ***khả dụng*** [reachable] từ u thông qua p, mà đôi lúc ta viết là $u \overset{p}{\leadsto} u'$ nếu G có hướng. Một lộ trình là ***đơn giản*** [simple] nếu tất cả các đỉnh trong lộ trình là riêng biệt. Trong Hình 5.2(a), lộ trình <1, 2, 5, 4> là một lộ trình đơn giản có chiều dài 3. Lộ trình <2, 5, 4, 5> không đơn giản.

Một ***lộ trình con*** [subpath] của lộ trình $p = <v_0, v_1, ..., v_k>$ là một dãy con tiếp giáp các đỉnh của nó. Nghĩa là, với bất kỳ $0 \leq i \leq j \leq k$, dãy con các đỉnh $<v_i, v_{i+1}, ..., v_j>$ là một lộ trình con của p.

Trong một đồ thị có hướng, một lộ trình $<v_0, v_1, ..., v_k>$ hình thành một ***chu trình*** [cycle] nếu $v_0 = v_k$ và lộ trình chứa ít nhất một cạnh. Chu trình là ***đơn giản*** [simple] nếu, thêm vào đó, $v_1, v_2, ..., v_k$ là riêng biệt. Một vòng tự lặp là một chu trình có chiều dài 1. Hai lộ trình $<v_0, v_1, v_2, ..., v_{k-1}, v_0>$ và $<v'_0, v'_1, v'_2, ..., v'_{k-1}, v'_0>$ hình thành cùng chu trình nếu ở đó tồn tại một số nguyên j sao cho $v'_i = v_{(i+j) \bmod k}$ với $i = 0, 1, ..., k - 1$. Trong Hình 5.2(a), lộ trình <1,2,4, 1> hình thành cùng chu trình như các lộ trình <2, 4, 1, 2> và <4, 1, 2, 4>. Chu trình này là đơn giản, nhưng chu trình <1, 2, 4, 5, 4, 1> thì không. Chu trình <2, 2> được hình thành bởi cạnh <2, 2> là một vòng tự lặp. Một đồ thị có hướng mà không có các vòng tự lặp là ***đơn giản***. Trong một đồ thị không có hướng, một lộ trình $<v_0, v_1, ..., v_k>$ hình thành một ***chu trình*** nếu $v_0 = v_k$ và $v_1, v_2, ..., v_k$ là riêng biệt. Ví dụ, trong Hình 5.2(b), lộ trình <1, 2, 5, 1> là một chu trình. Một đồ thị không có các chu trình sẽ là ***phi xiclic*** (hoặc phi chu trình = acyclic).

Một đồ thị không có hướng sẽ ***liên thông*** [connected] nếu mọi cặp các đỉnh được nối bởi một lộ trình. Các thành phần liên thông của một đồ thị là các lớp tương đương các đỉnh dưới quan hệ "là khả dụng từ"

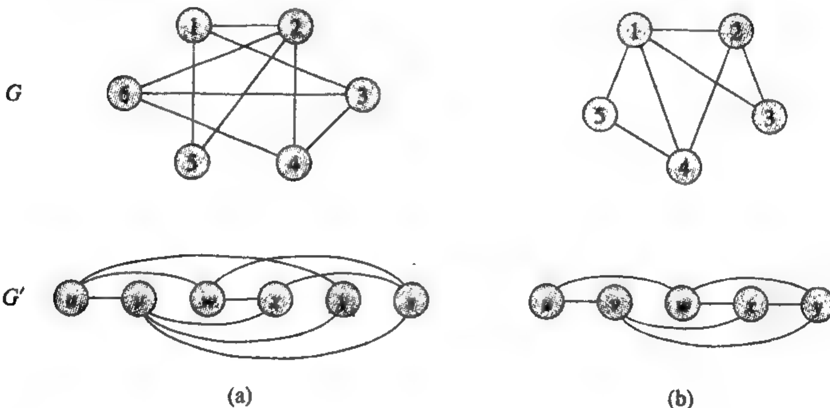
["is reachable from"]. Đồ thị trong Hình 5.2(b) có ba thành phần liên thông: {1, 2, 5}, {3, 6}, và {4}. Mọi đỉnh trong {1, 2, 5} là khả dụng từ mọi đỉnh khác trong {1, 2, 5}. Một đồ thị không có hướng được liên thông nếu nó có chính xác một thành phần liên thông, nghĩa là, nếu mọi đỉnh là khả dụng từ mọi đỉnh khác.

Một đồ thị có hướng được ***liên thông chặt*** [strongly connected] nếu mọi hai đỉnh đều khả dụng với nhau. Các ***thành phần liên thông chặt*** của một đồ thị là các lớp tương đương của các đỉnh dưới quan hệ "là khả dụng tương hỗ" . Một đồ thị có hướng được liên thông chặt nếu nó chỉ có một thành phần liên thông chặt. Đồ thị trong Hình 5.2(a) có ba thành phần liên thông chặt: {1, 2, 4, 5}, {3}, và {6}. Tất cả các cặp của các đỉnh trong {1, 2, 4, 5} đều khả dụng tương hỗ. Các đỉnh {3, 6} không hình thành một thành phần liên thông chặt, bởi đỉnh 6 không khả dụng từ đỉnh 3.

Hai đồ thị $G = (V, E)$ và $G' = (V', E')$ là ***đẳng cấu*** [isomorphic] nếu ở đó tồn tại một phép song ánh $f : V \rightarrow V'$ sao cho $(u, v) \in E$ nếu và chỉ nếu $(f(u), f(v)) \in E'$. Nói cách khác, ta có thể gán nhãn lại các đỉnh của G thành các đỉnh của G', duy trì các cạnh tương ứng trong G và G'. Hình 5.3(a) nêu một cặp đồ thị đẳng cấu G và G' có các tập hợp đỉnh tương ứng $V = \{1, 2, 3, 4, 5, 6\}$ và $V' = \{u, v, w, x, y, z\}$. Phép ánh xạ từ V đến V' được cung cấp bởi $f(1) = u$, $f(2) = v$, $f(3) = w$, $f(4) = x$, $f(5) = y$, $f(6) = z$ sẽ là hàm song ánh bắt buộc. Đồ thị trong Hình 5.3(b) không phải là đẳng cấu. Mặc dù cả hai đồ thị có 5 đỉnh và 7 cạnh, đồ thị đầu có một đỉnh có độ 4 và đồ thị dưới thì không.

Ta nói rằng của đồ thị $G' = (V', E')$ là một ***đồ thị con*** [subgraph] của $G = (V, E)$ nếu $V' \subseteq V$ và $E' \subseteq E$. Cho một tập hợp $V' \subseteq V$, đồ thị con của ***G được cảm sinh*** bởi V' là đồ thị $G' = (V', E')$, ở đó

$$E' = \{(u,v) \in E : u,v \in V'\}$$



Hình 5.3 (a) Một cặp đồ thị đẳng cấu. Các đỉnh của đồ thị đầu được ánh xạ theo các đỉnh của đồ thị dưới bởi $f(1) = u$, $f(2) = v$, $f(3) = w$, $f(4) = x$, $f(5) = y$, $f(6) = z$. **(b)** Hai đồ thị không đẳng cấu, bởi đồ thị đầu có một đỉnh có độ 4 và đồ thị dưới thì không.

Đồ thị con được cảm sinh bởi tập hợp đỉnh {1, 2, 3, 6} trong Hình 5.2 (a) xuất hiện trong Hình 5.2 (c) và có tập hợp cạnh {(1, 2), (2, 2), (6, 3)}.

Cho một đồ thị không có hướng $G = (V, E)$, ***phiên bản có hướng*** của G là đồ thị có hướng $G' = (V, E')$, ở đó $(u, v) \in E'$ nếu và chỉ nếu $(u, v) \in E$. Nghĩa là, từng cạnh không có hướng (u, v) trong G được thay trong phiên bản có hướng bằng hai cạnh có hướng (u, v) và (v, u). Cho một đồ thị có hướng $G = (V, E)$, ***phiên bản không có hướng*** của G là đồ thị không có hướng $G' = (V, E')$, ở đó $(u, v) \in E'$ nếu và chỉ nếu $u \neq v$ và $(u, v) \in E$. Nghĩa là, phiên bản không có hướng chứa các cạnh của G "có các hướng của chúng được gỡ bỏ" và có các vòng tự lặp được loại bỏ. (Bởi (u, v) và (v, u) là cùng một cạnh trong một đồ thị không có hướng, phiên bản không có hướng của một đồ thị có hướng chứa nó chỉ một lần, cho dù đồ thị có hướng chứa cả hai cạnh (u, v) và (v, u).) Trong một đồ thị có hướng $G = (V, E)$, một ***láng giềng*** [neighbor] của một đỉnh u là một đỉnh bất kỳ kề với u trong phiên bản không có hướng của G. Nghĩa là, v là một láng giềng của u nếu hoặc $(u, v) \in E$ hoặc $(v, u) \in E$. Trong một đồ thị không có hướng, u và v là những láng giềng nếu chúng kề nhau.

Có vài kiểu đồ thị được gán các tên đặc biệt. Một ***đồ thị đầy đủ*** [complete graph] là một đồ thị không có hướng ở đó mọi cặp các đỉnh kề nhau. Một ***đồ thị hai nhánh*** [bipartite graph] là một đồ thị không có hướng $G = (V, E)$ ở đó V có thể được phân hoạch thành hai tập hợp V_1 và V_2 sao cho $(u, v) \in E$ hàm ý hoặc $u \in V_1$ và $v \in V_2$ hoặc $u \in V_2$ và $v \in V_1$. Nghĩa là, tất cả các cạnh rơi giữa hai tập hợp V_1 và V_2. Một đồ thị không có hướng, phi chu trình, là một ***rừng*** [forest], và một đồ thị không có hướng, phi chu trình, liên thông, là một ***cây (tự do)*** (xem Đoạn 5.5). Người ta thường lấy các mẫu tự đầu của cụm từ "directed acyclic graph" [đồ thị có hướng phi chu trình] để gọi một đồ thị như vậy là ***dag***.

Có hai biến thể của đồ thị mà thỉnh thoảng ta có thể gặp. Một ***đa đồ thị*** [multigraph] giống như một đồ thị không có hướng, nhưng nó có thể có cả nhiều cạnh giữa các đỉnh lẫn các vòng tự lặp. Một ***quảng đồ*** [hypergraph] giống như một đồ thị không có hướng, nhưng thay vì nối hai đỉnh mỗi ***quảng cạnh*** [hyperedge] nối một tập hợp con các đỉnh tùy ý. Nhiều thuật toán viết cho đồ thị có hướng và không có hướng bình thường có thể được thích ứng để chạy trên các cấu trúc kiểu đồ thị này.

Bài tập

5.4-1

Những người tham dự buổi tiệc của khoa bắt tay chào nhau, và mỗi

Giáo sư nhớ họ đã bắt tay bao nhiêu lần. Cuối buổi tiệc, khoa trưởng cộng cả thảy số lần mà mỗi giao sư bắt tay. Chứng tỏ kết quả là chẵn bằng cách chứng minh ***bổ đề bắt tay***: nếu $G = (V, E)$ là một đồ thị không có hướng, thì

$$\sum_{v \in V} \text{degree}(v) = 2|E|$$

5.4-2

Chứng tỏ trong một đồ thị không có hướng, chiều dài của một chu trình phải ít nhất là 3.

5.4-3

Chứng tỏ nếu một đồ thị có hướng hoặc không có hướng chứa một lộ trình giữa hai đỉnh u và v, thì nó chứa một lộ trình đơn giản giữa u và v. Chứng tỏ nếu một đồ thị có hướng chứa một chu trình, thì nó chứa một chu trình đơn giản.

5.4-4

Chứng tỏ bất kỳ đồ thị không có hướng, liên thông $G = (V, E)$ thỏa $|E| \geq |V| - 1$.

5.4-5

Xác minh trong một đồ thị không có hướng, quan hệ "là khả đụng từ" ["is reachable from"] là một quan hệ tương đương trên các đỉnh của đồ thị. Tính chất nào trong số ba tính chất của một quan hệ tương đương áp dụng chung cho quan hệ "là khả đụng từ" trên các đỉnh của một đồ thị có hướng?

5.4-6

Nêu phiên bản không có hướng của đồ thị có hướng trong Hình 5.2(a)? Nêu phiên bản có hướng của đồ thị không có hướng trong Hình 5.2(b)?

5.4-7 *

Chứng tỏ một quảng đồ [hypergraph] có thể được biểu thị bằng một đồ thị hai nhánh nếu ta để sự liên thuộc trong quảng đồ tương ứng với sự kệ cận trong đồ thị hai nhánh. (*Mách nước:* Cho một tập hợp các đỉnh trong đồ thị hai nhánh tương ứng với các đỉnh của quảng đồ, và để tập hợp các đỉnh kia của đồ thị hai nhánh tương ứng với các quảng cạnh [hyperedges].)

5.5 Cây

Giống như đồ thị, có nhiều khái niệm về cây tuy có liên quan, nhưng hơi khác nhau. Đoạn này trình bày các phần định nghĩa và các tính chất toán học của vài kiểu cây. Các đoạn 11.4 và 23.1 mô tả cách biểu thị các cây trong một bộ nhớ máy tính.

5.5.1 Cây tự do

Như đã định nghĩa trong Đoạn 5.4, một ***cây tự do*** [free tree] là một đồ thị không có hướng, phi chu trình, và liên thông. Ta thường bỏ qua tính từ "tự do" khi nói đồ thị là một cây. Nếu một đồ thị không có hướng là phi chu trình nhưng có thể gián đoạn, nó là một ***rừng*** [forest]. Nhiều thuật toán áp dụng cho các cây cũng áp dụng cho các rừng. Hình 5.4(a) có nêu một cây tự do, và Hình 5.4(b) có nêu một rừng. Rừng trong Hình 5.4(b) không phải là một cây bởi nó không liên thông. Đồ thị trong Hình 5.4(*c*) không phải là cây cũng không phải là rừng, bởi nó chứa một chu trình.

Định lý dưới đây chốt giữ nhiều sự việc quan trọng về các cây tự do.

Định lý 5.2 (Các tính chất của các cây tự do)

Cho $G = (V, E)$ là một đồ thị không có hướng. Các phát biểu dưới đây là tương đương.

1. G là một cây tự do.

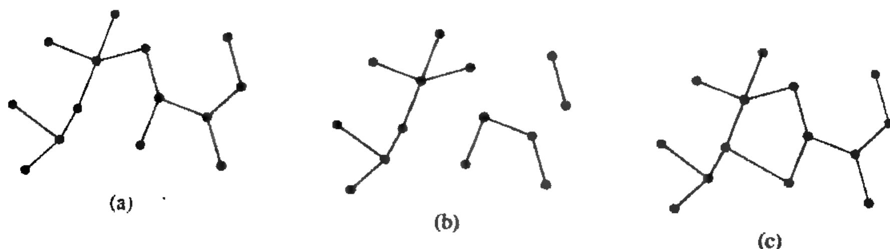
2. Hai đỉnh bất kỳ trong G được nối bởi một lộ trình đơn giản duy nhất.

3. G được liên thông, nhưng nếu gỡ bỏ một cạnh bất kỳ ra khỏi E, đồ thị kết quả sẽ gián đoạn.

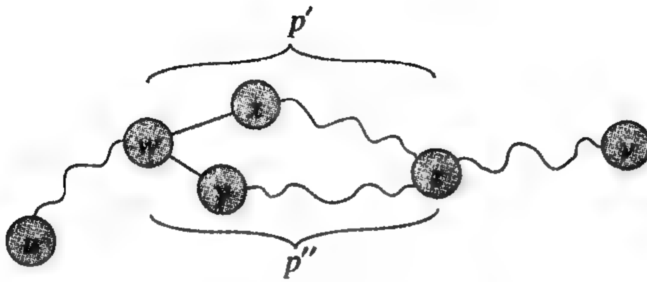
4. G được liên thông, và $|E| = |V| - 1$.

5. G là phi chu trình, và $|E| = |V| - 1$.

6. G là phi chu trình, nhưng nếu bổ sung một cạnh bất kỳ vào E, đồ thị kết quả sẽ chứa một chu trình.



Hình 5.4 (a) Một cây tự do. **(b)** Một rừng. **(c)** Một đồ thị chứa một chu trình và do đó không phải là một cây hay một rừng.



Hình 5.5 Một bước trong phép chứng minh của Định lý 5.2: nếu (1) G là một cây tự do, thì (2) hai đỉnh bất kỳ trong G được liên thông bởi một lộ trình đơn giản duy nhất. Vì sự mâu thuẫn, ta giả sử các đỉnh u và v được liên thông bởi hai lộ trình đơn giản riêng biệt p_1 và p_2. Các lộ trình này trước tiên phân kỳ tại đỉnh w, và trước tiên chúng tái hội tụ tại đỉnh z. Lộ trình p' được ghép nối với nghịch đảo của lộ trình p'' để hình thành một chu trình, cho ra sự mâu thuẫn.

Chứng minh (1) ⇒ (2): Bởi một cây liên thông, nên hai đỉnh bất kỳ trong G được nối bởi ít nhất một lộ trình đơn giản. Cho u và v là các đỉnh được liên thông bởi hai các lộ trình đơn giản riêng biệt p_1 và p_2, như đã nêu trong Hình 5.5. Cho w là đỉnh ở đó các lộ trình phân kỳ trước tiên; nghĩa là, w là đỉnh đầu tiên trên cả p_1 lẫn p_2 có phần tử kế vị [successor] trên p_1 là x và có phần tử kế vị trên p_2 là y, ở đó $x \neq y$. Cho z là đỉnh đầu tiên ở đó các lộ trình tái hội tụ; nghĩa là, z là đỉnh đầu tiên theo sau w trên p_1 cũng nằm trên p_2. Cho p' là lộ trình con của p_1 từ w qua x đến z, và cho p'' là lộ trình con của p_2 từ w qua y đến z. Các lộ trình p' và p'' không chia sẻ đỉnh nào ngoại trừ các điểm cuối của chúng. Như vậy, lộ trình có được bằng cách ghép nối p' và nghịch đảo của p'' là một chu trình. Đây là một mâu thuẫn. Như vậy, nếu G là một cây, ta có thể có tối đa một lộ trình giữa hai đỉnh.

(2) ⇒ (3): Nếu hai đỉnh bất kỳ trong G được liên thông bằng một lộ trình đơn giản duy nhất, thì G được liên thông. Cho (u, v) là cạnh bất kỳ trong E. Cạnh này là một lộ trình từ u đến v, và do đó nó phải là lộ trình duy nhất từ u đến v. Nếu gỡ bỏ (u, v) ra khỏi G, ta sẽ không có lộ trình nào từ u đến v, và như vậy việc gỡ bỏ nó sẽ làm gián đoạn G.

(3) ⇒ (4): Theo giả thiết, đồ thị G được liên thông, và theo Bài tập 5.4-4, ta có $|E| \geq |V| -1$. Ta sẽ chứng minh $|E| \leq |V| -1$ bằng phương pháp quy nạp. Một đồ thị liên thông với $n = 1$ hoặc $n = 2$ đỉnh sẽ có $n - 1$ cạnh. Giả sử rằng G có $n \geq 3$ đỉnh và tất cả các đồ thị thỏa (3) với ít hơn n đỉnh cũng thỏa $|E| \leq |V| - 1$. Việc gỡ bỏ một cạnh tùy ý ra khỏi G sẽ tách biệt đồ thị thành $k \geq 2$ thành phần liên thông (thực tế $k = 2$). Mỗi thành phần thỏa (3), nếu không G sẽ không thỏa (3). Như vậy,

bằng phương pháp quy nạp, số lượng các cạnh trong tất cả các thành phần tổ hợp sẽ tối đa là $|V| - k \le |V| - 2$. Cộng vào cạnh đả gỡ bỏ cho ra $|E| \le |V| - 1$.

(4) ⇒ (5): Giả sử G được liên thông và $|E| = |V| - 1$. Ta phải chứng tỏ G là phi chu trình. Giả sử G có một chu trình chứa k đỉnh $v_1, v_2, \ldots, v_k$. Cho $G_k = (V_k, E_k)$ là đồ thị con của G bao gồm chu trình. Lưu ý, $|V_k| = |E_k| = k$. Nếu $k < |V|$, ta phải có một đỉnh $v_{k+1} \in V - V_k$ kề với một đỉnh $v_i \in V_k$ nào đó, bởi G liên thông. Định nghĩa $G_{k+1} = (V_{k+1}, E_{k+1})$ là đồ thị con của G với $V_{k+1} = V_k \cup \{v_{k+1}\}$ và $E_{k+1} = E_k \cup \{(v_i, v_{k+1})\}$. Lưu ý, $|V_{k+1}| = |E_{k+1}| = k+1$. Nếu $k+1 < n$, ta có thể tiếp tục, định nghĩa G_{k+2} theo cùng cách, vân vân, cho đến khi ta được $G_n = (V_n, E_n)$, ở đó $n = |V|$, $V_n = V$, và $|E_n| = |V_n| = |V|$.

Bởi G_n là một đồ thị con của G, ta có $E_n \subseteq E$, và như vậy $|E| \ge |V|$, mâu thuẫn với giả thiết $|E| = |V| - 1$. Như vậy, G là phi chu trình.

(5) ⇒ (6): Giả sử G là phi chu trình và $|E| = |V| - 1$. Cho k là số lượng các thành phần liên thông của G. Mỗi thành phần liên thông theo định nghĩa là một cây tự do, và bởi (1) hàm ý (5), nên tổng của tất cả các cạnh trong tất cả các thành phần liên thông của G là $|V| - k$. Bởi vậy, ta phải có $k = 1$, và G thực tế là một cây. Bởi (1) hàm ý (2), nên hai đỉnh bất kỳ trong G được liên thông bởi một lộ trình đơn giản duy nhất. Như vậy, cộng một cạnh bất kỳ vào G sẽ tạo một chu trình.

(6) ⇒ (1): Giả sử G là phi chu trình nhưng nếu cộng một cạnh bất kỳ vào E, một chu trình sẽ được tạo. Ta phải chứng tỏ G liên thông. Cho u và v là các đỉnh tùy ý trong G. Nếu u và v chưa kề nhau, việc cộng cạnh (u, v) sẽ tạo một chu trình ở đó tất cả các cạnh trừ (u, v) thuộc về G. Như vậy, có một lộ trình từ u đến v, và bởi u và v đã được chọn tùy ý, nên G liên thông.

5.5.2 Các cây có gốc và có thứ tự

Một ***cây có gốc*** [rooted tree] là một cây tự do ở đó một trong các đỉnh được phân biệt với các đỉnh khác. Đỉnh phân biệt đó được gọi là ***gốc*** của cây. Ta thường gọi một đỉnh của cây có gốc là ***nút*** node[2]. Hình 5.6(a) có nêu một cây có gốc trên một tập hợp 12 nút có gốc 7.

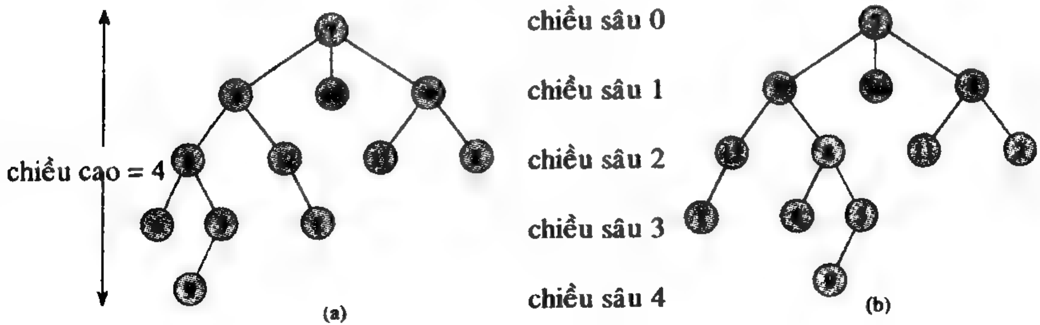
Xét một nút x trong một cây có gốc T với gốc r. Một nút bất kỳ y trên lộ trình duy nhất từ r đến x được gọi là ***tiền bối*** [ancestor] của x. Nếu y là một tiền bối của x, thì x là một hậu duệ của y. (Mọi nút đều vừa là một tiền bối vừa là một hậu duệ của chính nó.) Nếu y là một tiền bối của x và $x \ne y$, thì y là một ***tiền bối riêng*** của x và x là một ***hậu duệ riêng*** của y.

---

[2] Trong sách giáo khoa lý thuyết đồ thị, thuật ngữ "nút" thường được dùng đồng nghĩa với "đỉnh" [vertex]. Ta dành riêng thuật ngữ "nút" để chỉ một đỉnh của một cây có gốc.

Cây con có gốc tại x là cây được cảm sinh bởi các hậu duệ của *x*, có gốc tại *x*. Ví dụ, cây con có gốc tại nút 8 trong Hình 5.6(a) chứa các nút 8, 6, 5, và 9.

Nếu cạnh cuối trên lộ trình từ gốc *r* của một cây *T* đến một nút *x* là (*y*, *x*), thì *y* là ***cha*** [parent] của *x*, và *x* là một ***con*** [child] của *y*. Gốc là nút duy nhất trong *T* không có cha. Nếu hai nút có cùng cha, chúng được gọi là ***anh em ruột.*** Một nút không có các con là một ***nút ngoài*** [external node] hoặc ***lá.*** Một nút không lá là một ***nút trong*** [internal node].



Hình 5.6 Các cây có gốc và có thứ tự. **(a)** Một cây có gốc có chiều cao 4. Cây được vẽ theo cách chuẩn: gốc (nút 7) ở trên cùng, các con của nó (các nút có chiều sâu 1) nằm bên dưới nó, các con của các con đó (các nút có chiều sâu 2) nằm ngay bên dưới chúng, và vân vân. Nếu cây được sắp xếp thứ tự, thứ tự tương đối từ trái qua phải của các con của một nút là thứ tự có ý nghĩa; ngược lại là không. **(b)** Một cây khác có gốc. Là một cây có gốc, nó giống hệt như cây trong (a), nhưng là một cây có thứ tự, nó lại khác, bởi các con của nút 3 xuất hiện theo một thứ tự khác.

Số lượng con của một nút *x* trong một cây có gốc *T* được gọi là ***độ*** [degree] của *x*.[3]. Chiều dài của lộ trình từ gốc *r* đến một nút *x* là ***chiều sâu*** của *x* trong *T*. Chiều sâu lớn nhất của một nút bất kỳ trong *T* là ***chiều cao*** của *T*.

Một ***cây có thứ tự*** là một cây có gốc ở đó các con của mỗi nút được sắp xếp thứ tự. Nghĩa là, nếu một nút có *k* con, ta có một con đầu tiên, một con thứ hai, ..., và một con thứ *k*. Hai cây trong Hình 5.6 là khác nhau khi được xem là các cây có thứ tự, song là giống nhau khi được xem đơn thuần là các cây có gốc.

---

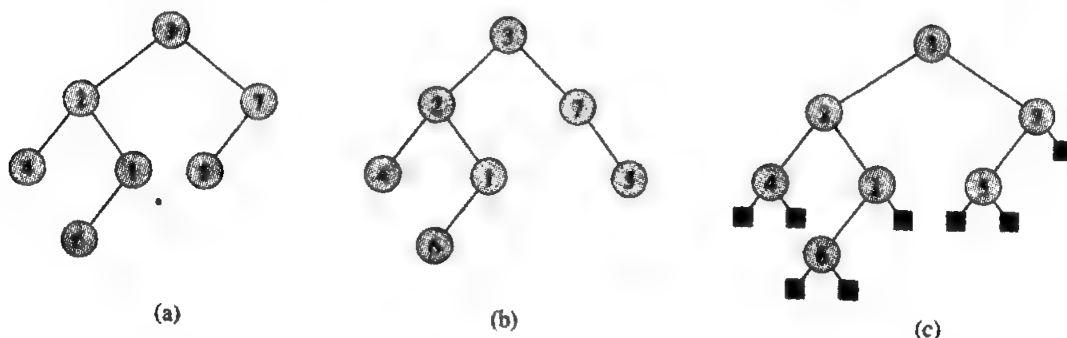
[3] Lưu ý, độ của một nút tùy thuộc vào việc *T* được xem là một cây có gốc hay một cây tự do. Cũng như trong bất kỳ đồ thị không có hướng nào, độ của một đỉnh trong một cây tự do là số lượng các đỉnh kề. Tuy nhiên, trong một cây có gốc, độ là số lượng các con—cha của một nút không đếm về phía độ của nó.

5.5.3 Các cây nhị phân và vị trí

Các cây nhị phân được mô tả thích hợp nhất theo đệ quy. Một ***cây nhị phân T*** là một cấu trúc được định nghĩa trên một tập hợp hữu hạn các nút hoặc

- không chứa nút nào, hoặc
- bao gồm ba tập hợp nút rời nhau: một nút ***gốc***, một cây nhị phân tên ***cây con trái*** của nó, và một cây nhị phân tên ***cây con phải*** của nó.

Cây nhị phân không chứa nút nào được gọi là ***cây trống*** hoặc ***cây rỗng***, đôi lúc được ký hiệu là NIL. Nếu cây con trái không trống, gốc của nó được gọi là ***con trái*** [left child] của gốc (của nguyên) cả cây. Cũng vậy, gốc của một cây con phải không rỗng là ***con phải*** [right child] của gốc cả cây. Nếu cây con là cây rỗng NIL, ta nói con bị ***vắng*** [absent] hoặc ***thiếu*** [missing]. Hình 5.7(a) có nêu một cây nhị phân.

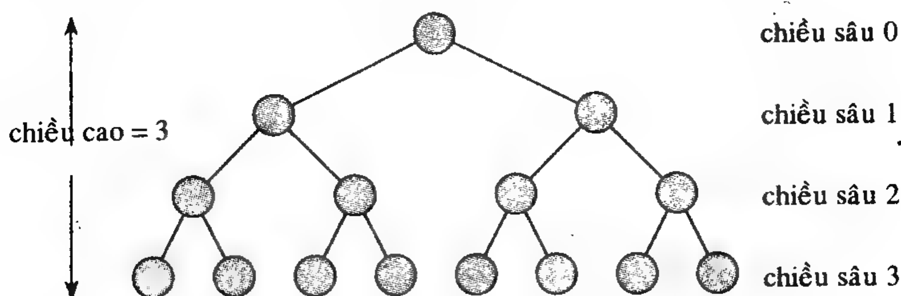


Hình 5.7 Các cây nhị phân. **(a)** Một cây nhị phân được vẽ theo cách chuẩn. Con trái của một nút được vẽ bên dưới nút và về bên trái. Con phải được vẽ bên dưới và về bên phải. **(b)** Một cây nhị phân khác với cây trong (a). Trong (a), con trái của nút 7 là 5 và con phải "vắng". Trong (b), con trái của nút 7 vắng và con phải là 5. Là các cây có thứ tự, các cây này giống nhau, nhưng nếu là các cây nhị phân, chúng sẽ khác biệt. **(c)** Cây nhị phân trong (a) được biểu thị bởi các nút trong của toàn cây nhị phân: một cây có thứ tự ở đó từng nút trong có độ là 2. Các lá trong cây được nêu dưới dạng các ô vuông.

Một cây nhị phân không đơn thuần là một cây có thứ tự ở đó mỗi nút có độ tối đa là 2. Ví dụ, trong một cây nhị phân, nếu một nút chỉ có một con, vị trí của con—dẫu là ***con trái*** hoặc ***con phải***—đều có ý nghĩa. Trong một cây có thứ tự, ta không phân biệt một con duy nhất là phải hay trái. Hình 5.7(b) có nêu một cây nhị phân khác với cây trong Hình 5.7(a) bởi vị trí của một nút. Tuy nhiên, với tư cách là các cây có thứ tự, hai cây giống nhau y hệt.

Để biểu thị thông tin định vị trong một cây nhị phân, ta có thể dùng các nút trong của một cây có thứ tự, như đã nêu trong Hình 5.7(c). Ý tưởng đó là thay mỗi con thiếu trong cây nhị phân bằng một nút không có con nào. Các nút lá này được vẽ dưới dạng các ô vuông trong hình. Cây kết quả là một ***cây nhị phân đầy đủ***: mỗi nút là một lá hoặc có độ chính xác là 2. Không có các nút độ-1. Bởi vậy, thứ tự của các con của một nút sẽ bảo toàn thông tin vị trí.

Thông tin định vị phân biệt các cây nhị phân với các cây có thứ tự có thể được khai triển thành các cây có trên 2 con mỗi nút. Trong một ***cây vị trí*** [positional tree], các con của một nút được gán nhãn bằng các số nguyên dương riêng biệt. Con thứ i của một nút là ***vắng*** nếu như không có con nào được gán nhãn bằng số nguyên i. Cây ***thuộc-k*** [k-ary] là một cây vị trí ở đó với mọi nút, tất cả các con có nhãn lớn hơn k đều thiếu. Như vậy, một cây nhị phân là một cây thuộc-k với $k = 2$.



Hình 5.8 Một cây nhị phân hoàn chỉnh có chiều cao 3 với 8 lá và 7 nút trong.

Một *cây* ***thuộc-k hoàn chỉnh*** là một cây thuộc-k ở đó tất cả các lá đều có cùng chiều sâu và tất cả các nút trong đều có độ k. Hình 5.8 nêu một cây nhị phân hoàn chỉnh có chiều cao 3. Một cây thuộc-k hoàn chỉnh có chiều cao h sẽ có bao nhiêu lá? Gốc có k con tại chiều sâu 1, mỗi con có k con tại chiều sâu 2, vân vân. Như vậy, số lượng lá tại chiều sâu h là k^h. Bởi vậy, chiều cao của một cây thuộc-k hoàn chỉnh có n lá là $\log_k n$. Số lượng nút trong của một cây thuộc-k hoàn chỉnh có chiều cao h sẽ là

$$1 + k + k^2 + \dots + k^{h-1} = \sum_{i=0}^{h-1} k^i$$
$$= \frac{k^h - 1}{k - 1}$$

theo phương trình (3.3). Như vậy, một cây nhị phân hoàn chỉnh có $2^h - 1$ nút trong.

Bài tập

5.5-1

Vẽ tất cả các cây tự do bao gồm 3 đỉnh *A*, *B*, và *C*. Vẽ tất cả các cây có gốc có các nút *A*, *B*, và *C* với *A* là gốc. Vẽ tất cả các cây có thứ tự có các nút *A*, *B*, và *C* với *A* là gốc. Vẽ tất cả các cây nhị phân có các nút *A*, *B*, và *C* với *A* là gốc.

5.5-2

Chứng tỏ với $n \geq 7$, ở đó tồn tại một cây tự do trên *n* nút sao cho việc lấy từng trong số *n* nút làm gốc sẽ dẫn đến một cây có gốc khác.

5.5-3

Cho $G = (V, E)$ là một đồ thị phi chu trình có hướng ở đó có một đỉnh $v_0 \in V$ sao cho ở đó tồn tại một lộ trình duy nhất từ v_0 đến mọi đỉnh $v \in V$. Chứng minh phiên bản không có hướng của *G* hình thành một cây.

5.5-4

Chứng tỏ bằng phương pháp quy nạp số lượng nút có độ-2 trong một cây nhị phân bất kỳ nhỏ hơn số lượng lá là 1.

5.5-5

Chứng tỏ bằng phương pháp quy nạp một cây nhị phân với *n* nút sẽ có chiều cao ít nhất là $\lfloor \lg n \rfloor$.

5.5-6 *

Chiều dài lộ trình trong [internal path length] của một cây nhị phân đầy đủ là tổng chiều sâu của mỗi nút, được lấy trên tất cả các nút trong cây. Cũng vậy, ***chiều dài lộ trình ngoài*** [external path length] là tổng chiều sâu của mỗi lá, được lấy trên tất cả các lá trong cây. Xét một cây nhị phân đầy đủ có *n* nút trong, chiều dài lộ trình trong *i*, và chiều dài lộ trình ngoài *e*. Chứng minh $e = i + 2n$.

5.5-7 *

Ta hãy liên hợp một "trọng số" [weight] $w(x) = 2^{-d}$ với mỗi lá *x* có chiều sâu *d* trong một cây nhị phân *T*. Chứng minh $\sum_x w(x) \leq 1$, ở đó tổng được lấy trên tất cả lá *x* trong *T*. (Điều này được gọi là ***bất đẳng thức Kraft***.)

5.5-8 *

Chứng tỏ mọi cây nhị phân có *L* lá chứa một cây con có giữa $L/3$ và $2L/3$ lá, kể cả các giá trị trên đây.

Các Bài Toán

5-1 Tô màu đồ thị

Tô màu-k [k-coloring] của một đồ thị không có hướng $G = (V, E)$ là một hàm $c : V \rightarrow \{0, 1, ..., k - 1\}$ sao cho $c(u) \neq c(v)$ với mọi cạnh $(u, v) \in E$. Nói cách khác, có số 0, 1,..., k - 1 biểu thị cho k màu, và các đỉnh kề phải có các màu khác nhau.

a. Chứng tỏ bất kỳ cây nào cũng có thể có 2-màu.

b. Chứng tỏ nội dung dưới đây là tương đương:

1. G là hai nhánh.

2. G có thể có 2-màu.

3. G không có các chu trình có chiều dài lẻ.

c. Cho d là độ cực đại của một đỉnh bất kỳ trong một đồ thị G. Chứng minh G có thể được tô màu bằng $d + 1$ màu.

d. Chứng tỏ nếu G có $O(|V|)$ cạnh, thì G có thể được tô màu với $O(\sqrt{|V|})$ màu.

5-2 Đồ thị thân thiện

Phát biểu lại các câu dưới đây theo dạng một định lý về đồ thị không có hướng, rồi chứng minh nó. Giả sử tình bạn là đối xứng chứ không phải là phản xạ.

a. Trong một nhóm bất kỳ $n \geq 2$ người, có hai người có cùng số lượng bạn bè trong nhóm.

b. Mọi nhóm sáu người chứa ba người bạn chung hoặc ba người lạ chung.

c. Bất kỳ nhóm người nào cũng có thể được phân hoạch thành hai nhóm con sao cho ít nhất phân nửa số bạn bè của từng người thuộc về nhóm con mà người đó không phải là một phần tử.

d. Nếu mọi người trong một nhóm là bạn của ít nhất phân nửa số người trong nhóm, thì nhóm có thể được chừa chỗ quanh một chiếc bàn sao cho mọi người đều được ngồi giữa hai người bạn.

5-3 Chia đôi các cây

Nhiều thuật toán chia để trị tính toán trên đồ thị yêu cầu chia đồ thị thành hai đồ thị con có kích cỡ gần bằng nhau bằng cách gỡ bỏ một lượng nhỏ các cạnh. Bài toán này thẩm tra các kiểu chia đôi các cây.

a. Chứng tỏ bằng cách gỡ bỏ chỉ một cạnh, ta có thể phân hoạch các đỉnh của một cây nhị phân n-đỉnh bất kỳ thành hai tập hợp A và B sao

cho $|A| \le 3n/4$ và

$|B| \le 3n/4$.

b. Chứng tỏ hằng 3/4 trong phần (a) là tối ưu trong trường hợp xấu nhất bằng cách nêu một ví dụ về một cây đơn giản có phân hoạch hầu như cân đều khi gỡ bỏ chỉ một cạnh sẽ có $|A| = 3n/4$.

c. Chứng tỏ bằng cách gỡ bỏ tối đa $O(\lg n)$ cạnh, ta có thể phân hoạch các đỉnh của một cây n-đỉnh bất kỳ thành hai tập hợp A và B sao cho $|A| = \lfloor n/2 \rfloor$ và $|B| = \lceil n/2 \rceil$.

Ghi chú Chương

G. Boole đã tiên phong trong việc phát triển lôgic ký hiệu, và đã giới thiệu nhiều hệ ký hiệu tập hợp cơ bản trong một cuốn sách xuất bản năm 1854. Lý thuyết tập hợp hiện đại đã được tạo bởi G. Cantor trong thời kỳ 1874-1895. Cantor đã tập trung chủ yếu vào các tập hợp bản số vô hạn. Thuật ngữ "hàm" [function] được quy cho G. W. Leibnitz, người đã dùng nó để chỉ vài kiểu công thức toán học. Định nghĩa hạn chế của ông đã được tổng quát hóa nhiều lần. Lý thuyết đồ thị bắt nguồn vào năm 1736, khi L. Euler chứng minh rằng không thể băng qua mỗi trong số bảy cây cầu trong thành phố Konigsberg chính xác một lần và trở về điểm bắt đầu.

Cuốn sách của Harary [94] có đề cập nhiều định nghĩa hữu ích và kết quả từ lý thuyết đồ thị.

6 Đếm và Xác Suất

Chương này ôn lại căn bản về toán tổ hợp và lý thuyết xác suất. Nếu đã có một kiến thức vững về các lĩnh vực này, bạn có thể lướt nhanh qua phần đầu chương và tập trung vào các đoạn sau. Hầu hết các chương không yêu cầu xác suất, nhưng với vài chương, nó là thiết yếu.

Đoạn 6.1 ôn lại các kết quả căn bản về lý thuyết đếm, kể cả các công thức chuẩn cho các phép hoán vị đếm và các tổ hợp. Đoạn 6.2 mô tả các tiên đề xác suất và các yếu tố cơ bản liên quan đến các phép phân phối xác suất. Các biến ngẫu nhiên được trình bày trong Đoạn 6.3, cùng với các tính chất của sự dự trù [expectation] và phương sai. Đoạn 6.4 nghiên cứu các phép phân phối nhị thức và theo cấp số nhân phát sinh từ việc nghiên cứu các lần thử Bernoulli. Đoạn 6.5 tiếp tục nghiên cứu phép phân phối nhị thức, một phần đề cập cao cấp về "mặt sấp" của phép phân phối. Cuối cùng, Đoạn 6.6 minh họa tiến trình phân tích xác suất thông qua ba ví dụ: nghịch lý ngày sinh nhật, ném bóng ngẫu nhiên vào các giỏ, và các vệt thắng.

6.1 Đếm

Lý thuyết đếm gắng trả lời câu hỏi "Bao nhiêu?" mà không thực tế điểm lại bao nhiêu. Ví dụ, ta có thể hỏi, "Có bao nhiêu số n-bit khác nhau?" hoặc "Có bao nhiêu kiểu sắp xếp n thành phần riêng biệt?" Trong đoạn này, ta ôn lại các thành phần của lý thuyết đếm. Do có vài nội dung mặc nhận một kiến thức căn bản về tập hợp, bạn đọc nên bắt đầu bằng việc ôn lại nội dung trong Đoạn 5.1.

Các quy tắc về tổng và tích

Một tập hợp các mục mà ta muốn đếm đôi lúc có thể được diễn tả dưới dạng một hợp [union] của tập hợp rời nhau hoặc dưới dạng một tích Đề các các tập hợp.

Quy tắc về tổng nói rằng số lượng cách chọn một thành phần từ một trong hai tập hợp *rời nhau* là tổng của các bản số [cardinalities] của các tập hợp. Nghĩa là, nếu A và B là hai tập hợp hữu hạn không có các phần tử chung, thì $|A \cup B| = |A| + |B|$, theo phương trình (5.3). Ví dụ, mỗi

vị trí trên biển số xe là một mẫu tự hoặc một chữ số. Do đó, số lượng các khả năng cho mỗi vị trí là 26 + 10 = 36, bởi có 26 chọn lựa nếu nó là một mẫu tự và 10 chọn lựa nếu nó là một chữ số.

Quy tắc về tích nói rằng số lượng cách chọn một cặp có thứ tự đó là số lượng cách chọn thành phần đầu tiên nhân với số lượng cách chọn thành phần thứ hai. Nghĩa là, nếu A và B là hai tập hợp hữu hạn, thì $|A \times B| = |A| \cdot |B|$, mà đơn giản là phương trình (5.4). Ví dụ, nếu một cửa tiệm bán kem cung cấp 28 hương vị kem và 4 kiểu lớp mặt, số lượng kem nước quả khả dĩ có một muỗng xúc kem và một lớp mặt là $28 \cdot 4 = 112$.

Chuỗi

Một ***chuỗi*** [string] trên một tập hợp hữu hạn S là một dãy các thành phần của S. Ví dụ, có 8 chuỗi nhị phân có chiều dài 3:

000, 001, 010, 011, 100, 101, 110, 111 .

Đôi lúc ta gọi một chuỗi có chiều dài k là một ***chuỗi-k*** [k-string]. Một ***chuỗi con s'*** của một chuỗi s là một dãy có thứ tự gồm các thành phần liên tục của s. Một ***chuỗi con-k*** của một chuỗi là một chuỗi con có chiều dài k. Ví dụ, 010 là một chuỗi con 3 của 01101001 (chuỗi con 3 bắt đầu tại vị trí 4), nhưng 111 không phải là một chuỗi con của 01101001.

Một chuỗi-k trên một tập hợp S có thể được xem như một thành phần của tích Đề các S^k của các bộ-k [k-tuples]; như vậy, có $|S|^k$ chuỗi có chiều dài k. Ví dụ, số lượng các chuỗi-k nhị phân là 2^k. Theo trực giác, để kiến tạo một chuỗi-k trên một tập hợp n, ta có n cách để lấy thành phần đầu tiên; với từng chọn lựa này, ta có n cách để lấy thành phần thứ hai; và vân vân k lần. Kiểu kiến tạo này dẫn đến tích gấp-k [k-fold] $n.n...n = n^k$ dưới dạng số lượng các chuỗi-k.

Phép hoán vị

Một *phép hoán vị* của một tập hợp hữu hạn S là một dãy có thứ tự gồm tất cả các thành phần của S, với mỗi thành phần xuất hiện chính xác một lần. Ví dụ, nếu $S = \{a, b, c\}$, có 6 phép hoán vị của S:

abc, acb, bac, bca, cab, cba .

Có $n!$ phép hoán vị của một tập hợp n thành phần, bởi thành phần đầu tiên của dãy có thể được chọn theo n cách, thành phần thứ hai theo $n - 1$ cách, thành phần thứ ba theo $n - 2$ cách, và vân vân.

Phép hoán vị-k của S là một dãy có thứ tự gồm k thành phần của S, mà không có thành phần nào xuất hiện nhiều hơn một lần trong dãy. (Như vậy, một phép hoán vị bình thường chẳng qua là một phép hoán vị-n của một tập hợp n.) Mười hai phép hoán vị-2 của tập hợp $\{a, b, c, d\}$ là

ab, ac, ad, ba, bc, bd, ca, cb, cd, da, db, dc .

Số lượng các phép hoán vị-*k* của một tập hợp *n* là

$$n(n - 1)(n - 2) \dots (n - k + 1) = \frac{n!}{(n-k)!}, \tag{6.1}$$

bởi có *n* cách chọn thành phần đầu tiên, *n* -1 cách chọn thành phần thứ hai, và vân vân cho đến khi *k* thành phần được chọn, thành phần cuối là một chọn lựa từ *n* - *k* + 1 thành phần.

Các tổ hợp

Một ***tổ hợp-k*** [*k*-combination] của một tập hợp-*n* *S* chẳng qua là một tập hợp con-*k* của *S*. Có sáu tổ hợp-2 của tập hợp-4 {*a, b, c, d* }:

ab, ac, ad, bc, bd, cd .

(Ở đây ta dùng lối ký hiệu tốc ký tập hợp-2 {*a, b*} bằng *ab,* và vân vân.) Ta có thể kiến tạo một tổ hợp-*k* của một tập hợp *n* bằng cách chọn *k* thành phần riêng biệt (khác nhau) từ tập hợp *n.*

Số lượng tổ hợp-*k* của một tập hợp *n* có thể được diễn tả theo dạng số lượng phép hoán vị-*k* của một tập hợp *n*. Với mọi tổ hợp-*k*, ta có chính xác *k*! phép hoán vị của các thành phần của nó, mỗi phép hoán vị là một phép hoán vị-*k* riêng biệt của tập hợp *n*. Như vậy, số lượng tổ hợp-*k* của một tập hợp *n* là số lượng phép hoán vị-*k* chia cho *k*!; từ phương trình (6.1), con số này là

$$\frac{n!}{k!\,(n-k)!}. \tag{6.2}$$

Với *k* = 0, công thức này cho ta biết số lượng cách chọn 0 thành phần từ một tập hợp *n* là 1 (chứ không phải 0), bởi 0! = 1.

Các hệ số nhị thức

Ta dùng hệ ký hiệu $\binom{n}{k}$ (đọc là "*n* chọn *k*") để thể hiện số lượng tổ hợp-*k* của một tập hợp *n*. Từ phương trình (6.2), ta có

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}. \tag{6.3}$$

Công thức này là đối xứng trong *k* và *n* - *k*:

$$\binom{n}{k} = \binom{n}{n-k}. \tag{6.4}$$

Các con số này cũng được xem là ***các hệ số nhị thức***, do dáng vẻ của chúng trong ***phép khai triển nhị thức***:

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}. \qquad (6.5)$$

Một trường hợp đặc biệt của phép khai triển nhị thức xảy ra khi $x = y = 1$

$$2^n = \sum_{k=0}^{n} \binom{n}{k} . \qquad (6.6)$$

Công thức này tương ứng với việc đếm 2^n chuỗi-n nhị phân theo số lượng các 1 mà chúng chứa: có $\binom{n}{k}$ chuỗi-n nhị phân chứa chính xác k 1, bởi có $\binom{n}{k}$ cách chọn k từ n vị trí ở đó đặt các 1.

Có nhiều đồng nhất thức liên quan đến các hệ số nhị thức. Các bài tập cuối đoạn này cho ta cơ hội chứng minh một số.

Các cận nhị thức

Đôi lúc ta cần định cận kích cỡ của một hệ số nhị thức. Với $1 \le k \le n$, ta có cận dưới

$$\begin{aligned}\binom{n}{k} &= \frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 1} \\ &= \left(\frac{n}{k}\right)\left(\frac{n-1}{k-1}\right)\cdots\left(\frac{n-k+1}{1}\right) \\ &\ge \left(\frac{n}{k}\right)^k . \qquad (6.7)\end{aligned}$$

Vật dụng bất đẳng thức $k! \ge (k/e)^k$ dẫn xuất từ công thức Stirling (2.12), ta được các cận trên

$$\begin{aligned}\binom{n}{k} &= \frac{n(n-1)\dots(n-k+1)}{k(k-1)\dots 1} \\ &\le \frac{n^k}{k!} \qquad (6.8) \\ &\le \left(\frac{en}{k}\right)^k . \qquad (6.9)\end{aligned}$$

Với tất cả $0 \le k \le n$, ta có thể dùng phương pháp quy nạp (xem Bài tập 6.1-12) để chứng minh cận

$$\binom{n}{k} \le \frac{n^n}{k^k (n-k)^{n-k}} , \qquad (6.10)$$

ở đó, để tiện dụng, ta mặc nhận rằng $0^0 = 1$. Với $k = \lambda n$, ở đó $0 \le A \le 1$, cận này có thể được viết lại là

$$\binom{n}{\lambda n} \le \frac{n^n}{(\lambda n)^{\lambda n}((1-\lambda)n)^{(1-\lambda)n}}$$

$$= \left(\left(\frac{1}{\lambda}\right)^{\lambda}\left(\frac{1}{1-\lambda}\right)^{1-\lambda}\right)^{n} \quad (6.11)$$

$$= 2^{n\,H(\lambda)}, \quad (6.12)$$

ở đó

$$H(\lambda) = -\lambda \lg \lambda - (1-\lambda)\lg(1-\lambda) \quad (6.13)$$

là ***hàm entropi (nhị phân)*** và ở đó, để tiện dụng, ta mặc nhận rằng 0 lg 0 = 0, sao cho $H(0) = H(1) = 0$.

Bài tập

6.1-1

Một chuỗi-n có bao nhiêu chuỗi con-k? (Xét các chuỗi con-k giống nhau tại các vị trí khác nhau như là khác nhau.) Một chuỗi-n có tổng cộng bao nhiêu chuỗi con?

6.1-2

Một ***hàm Bool*** đầu vào-n, kết xuất-m là một hàm từ $\{\text{TRUE, FALSE}\}^n$ đến $\{\text{TRUE, FALSE}\}^m$. Có bao nhiêu hàm Bool đầu vào-n, kết xuất-1? Có bao nhiêu hàm Bool đầu vào-n, kết xuất-m?

6.1-3

Có bao nhiêu cách để n Giáo sư có thể ngồi quanh một bàn tròn hội nghị? Xét hai kiểu xếp ghế là như nhau nếu kiểu này có thể quay để tạo thành kiểu kia.

6.1-4

Có bao nhiêu cách để có thể chọn ba con số riêng biệt từ tập hợp {1, 2,...,100} sao cho tổng của chúng là chẵn?

6.1-5

Chứng minh đồng nhất thức

$$\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1} \quad (6.14)$$

với $0 < k \le n$.

6.1-6

Chứng minh đồng nhất thức

$$\binom{n}{k} = \frac{n}{n-k}\binom{n-1}{k}$$

với $0 \leq k < n$.

6.1-7

Để chọn k đối tượng từ n, bạn có thể tạo một trong số các đối tượng đó thành khác biệt rồi xét xem đối tượng khác biệt đó có được chọn hay không. Dùng cách tiếp cận này để chứng minh rằng

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}.$$

6.1-8

Dùng kết quả của Bài tập 6.1-7, tạo một bảng với $n = 0,1,\ldots, 6$ và $0 \leq k \leq n$ của các hệ số nhị thức $\binom{n}{k}$ with $\binom{0}{0}$ ở trên cùng, $\binom{1}{0}$ và $\binom{1}{1}$ trên dòng kế tiếp, và vân vân. Một bảng các hệ số nhị thức như vậy được gọi là ***tam giác Pascal***.

6.1-9

Chứng minh

$$\sum_{i=0}^{n} i = \binom{n+1}{2}.$$

6.1-10

Chứng tỏ với bất kỳ $n \geq 0$ và $0 \leq k \leq n$, ta được giá trị cực đại [maximum value] của $\binom{n}{k}$ khi $k = \lfloor n/2 \rfloor$ hoặc $k = \lceil n/2 \rceil$.

6.1-11 *

Biện luận với bất kỳ $n \geq 0, j \geq 0, k \geq 0$, và $j + k \leq n$,

$$\binom{n}{j+k} \leq \binom{n}{j}\binom{n-j}{k}. \tag{6.15}$$

Nêu cả hai: một chứng minh đại số và một biện luận dựa trên một phương pháp để chọn $j + k$ mục từ n. Nêu một ví dụ ở đó không áp dụng đẳng thức.

6.1-12 *

Dùng phương pháp quy nạp trên $k \leq n/2$ để chứng minh bất đẳng thức (6.10), và dùng phương trình (6.4) để khai triển nó cho tất cả $k \leq n$.

6.1-13 *

Dùng phép xấp xỉ Stirling để chứng minh

$$\binom{2n}{n} = \frac{2^{2n}}{\sqrt{\pi n}}(1 + O(1/n)). \tag{6.16}$$

6.1-14 *

Bằng cách lấy vi phân hàm entropy $H(\lambda)$, chứng tỏ ta được giá trị cực đại của nó tại $\lambda = 1/2$. $H(1/2)$ là gì?

6.2 Xác suất

Xác suất là một công cụ thiết yếu để thiết kế và phân tích các thuật toán có tính xác suất và ngẫu nhiên. Đoạn này ôn lại căn bản về lý thuyết xác suất.

Ta định nghĩa xác suất theo dạng một ***không gian mẫu*** [sample space] S, là một tập hợp có các thành phần gọi là các ***sự kiện cơ bản*** [elementary events]. Mỗi sự kiện cơ bản có thể được xem như là một kết quả khả dĩ của một thí nghiệm. Với thí nghiệm tung hai đồng tiền phân biệt được, ta có thể xem không gian mẫu là có bao gồm tập hợp tất cả các chuỗi-2 khả dĩ trên {H, T}:

$S = \{HH, HT, TH, TT\}$.

Một ***sự kiện*** [event] là một tập hợp con[1] của không gian mẫu S. Ví dụ, trong thí nghiệm tung hai đồng tiền, sự kiện có được một mặt ngửa và một mặt sấp là {HT, TH}. Sự kiện S được gọi là ***sự kiện chắc chắn***, và sự kiện Ø? được gọi là ***sự kiện rỗng*** [hoặc sự kiện có xác suất không = null event]. Ta nói hai sự kiện A và B ***loại trừ nhau*** nếu $A \cap B = \emptyset$. Đôi lúc ta xem một sự kiện cơ bản $s \in S$ như sự kiện $\{s\}$. Theo định nghĩa, tất cả các sự kiện cơ bản là loại trừ nhau.

Các tiên đề của xác suất

Một ***phép phân phối xác suất*** Pr { } trên một không gian mẫu S là một phép ánh xạ từ các sự kiện của S theo các số thực sao cho các ***tiên đề xác suất*** dưới đây được thỏa:

1. $\Pr\{A\} \geq 0$ với bất kỳ sự kiện A.

2. $\Pr\{S\} = 1$.

3. $\Pr\{A \cup B\} = \Pr\{A\} + \Pr\{B\}$ với bất kỳ hai sự kiện loại trừ nhau A

[1] Với một phép phân phối xác suất chung, có thể có vài tập hợp con của không gian mẫu S không được xem là các sự kiện. Tình huống này thường nảy sinh khi không gian mẫu là vô hạn bất khả đếm. Yêu cầu chính đó là tập hợp các sự kiện của một không gian mẫu được khép kín dưới các phép toán lấy phần bù của một sự kiện, tạo thành hợp của một số lượng sự kiện hữu hạn hoặc đếm được, và lấy giao của một số lượng sự kiện hữu hạn hoặc đếm được. Phần lớn các phép phân phối xác suất mà ta sẽ gặp đều dựa trên các không gian mẫu hữu hạn hoặc đếm được, và ta thường xem tất cả các tập hợp con của một không gian mẫu là các sự kiện. Một ngoại lệ đáng lưu ý đó là phép phân phối xác suất đều liên tục, sẽ được trình bày ngắn gọn dưới đây.

và B. Chung hơn, với bất kỳ dãy (hữu hạn hoặc vô hạn khả đếm) các sự kiện $A_1, A_2, \ldots$ loại trừ nhau từng cặp,

$$\Pr\left\{\bigcup_i A_i\right\} = \sum_i \Pr\{A_i\}.$$

Ta gọi Pr $\{A\}$ là ***xác suất*** của sự kiện A. Ở đây ta lưu ý rằng tiên đề 2 là một yêu cầu chuẩn tắc hóa [normalization]: thực sự không có gì căn bản về chọn 1 làm xác suất của sự kiện chắc chắn, ngoại trừ nó là tự nhiên và tiện dụng.

Có vài kết quả theo ngay sau các tiên đề này và lý thuyết tập hợp căn bản (xem Đoạn 5.1). Sự kiện rỗng $\emptyset$ có xác suất $\Pr\{\emptyset\} = 0$. Nếu $A \subseteq B$, thì $\Pr\{A\} \le \Pr\{B\}$. Dùng $\overline{A}$ để thể hiện sự kiện $S - A$ (***phần bù*** của A), ta có $\Pr\{\overline{A}\} = 1 - \Pr\{A\}$. Với hai sự kiện A và B bất kỳ,

$$\Pr\{A \cup B\} = \Pr\{A\} + \Pr\{B\} - Pr\{A \cap B\} \qquad (6.17)$$

$$\le \Pr\{A\} + \Pr\{B\}. \qquad (6.18)$$

Trong ví dụ tung đồng tiền, giả sử rằng mỗi trong số bốn sự kiện cơ bản có xác suất là 1/4. Như vậy, xác suất để có ít nhất một mặt ngửa là

$$\Pr\{\text{HH, HT, TH}\} = \Pr\{\text{HH}\} + \Pr\{\text{HT}\} + \Pr\{\text{TH}\} = 3/4.$$

Một cách khác, do xác suất để có đúng ít hơn một mặt ngửa là $\Pr\{\text{TT}\} = 1/4$, nên xác suất để có ít nhất một mặt ngửa là $1 - 1/4 = 3/4$.

Các phép phân phối xác suất rời rạc

Một phép phân phối xác suất là ***rời rạc*** [discrete] nếu nó được định nghĩa trên một không gian mẫu hữu hạn hoặc vô hạn khả đếm. Cho S là không gian mẫu. Vậy với bất kỳ sự kiện A,

$$\Pr\{A\} = \sum_{s \in A} \Pr\{s\},$$

bởi các sự kiện cơ bản, nhất là những sự kiện trong A, loại trừ nhau. Nếu S là hữu hạn và mọi sự kiện cơ bản $s \in S$ có xác suất

$$\Pr\{s\} = 1/\{S\},$$

thì ta có ***phép phân phối xác suất đều*** trên S. Trong trường hợp như vậy, thí nghiệm thường được mô tả là "lấy một thành phần của S theo ngẫu nhiên."

Để lấy ví dụ, hãy xét tiến trình tung một ***đồng tiền cân*** [fiar coin], là đồng tiền mà xác suất có được một mặt ngửa cũng giống như xác suất có được một mặt sấp, nghĩa là, 1/2. Nếu tung đồng tiền n lần, ta có phép phân phối xác suất đều được định nghĩa trên không gian mẫu $S = \{$H,

T}^{n}, một tập hợp có kích cỡ 2^n. Mỗi sự kiện cơ bản trong S đều có thể biểu thị dưới dạng một chuỗi có chiều dài n trên $\{H, T\}$, và mỗi sự kiện xảy ra với xác suất $1/2^n$. Sự kiện

A = {chính xác k mặt ngửa và chính xác $n - k$ mặt sấp xảy ra}

là một tập hợp con của S có kích cỡ $|A| = \binom{n}{k}$, bởi có (k) chuỗi có chiều dài n trên {H, T} chứa chính xác k H'S. Như vậy xác suất của sự kiện A là Pr $\{A\} = \binom{n}{k}/2^n$.

Phép phân phối xác suất đều liên tục

Phép phân phối xác suất đều liên tục là một ví dụ của một phép phân phối xác suất ở đó tất cả các tập hợp con của không gian mẫu không được xem là các sự kiện. Phép phân phối xác suất đều liên tục được định nghĩa trên một khoảng đóng $[a, b]$ các số thực [reals], ở đó $a < b$. Theo trực giác, ta muốn từng điểm trong khoảng $[a, b]$ "hầu như bằng nhau." Tuy nhiên, có một số lượng điểm không đếm được, do đó nếu gán cho tất cả các điểm cùng xác suất dương, hữu hạn, ta không thể cùng lúc thỏa các tiên đề 2 và 3. Vì lý do này, ta chỉ muốn kết hợp một xác suất với *vài* tập hợp con của S sao cho các tiên đề được thỏa với các sự kiện này.

Với bất kỳ khoảng đóng $[c, d]$, ở đó $a \le c \le d \le b$, ***phép phân phối xác suất đều liên tục*** định nghĩa xác suất của sự kiện $[c, d]$ sẽ là

$$\Pr\{[c, d]\} = \frac{d - c}{b - a} .$$

Lưu ý, với một điểm $x = [x, x]$ bất kỳ, xác suất của x là 0. Nếu gỡ bỏ các điểm cuối của một khoảng $[c, d]$, ta được khoảng mở (c, d). Bởi $[c, d] = [c, c] \cup (c, d) \cup [d, d]$, tiên đề 3 cho ta $\Pr\{[c, d]\} = \Pr\{(c, d)\}$. Nói chung, tập hợp các sự kiện cho phép phân phối xác suất đều liên tục là bất kỳ tập hợp con nào của $[a, b]$ có thể có được bằng một hợp hữu hạn hoặc đếm được của các khoảng đóng và mở.

Xác suất có điều kiện và sự độc lập

Đôi lúc ta biết trước được phần nào về kết quả của một thí nghiệm. Ví dụ, giả sử một người bạn đã tung hai đồng tiền cân và đã nói cho bạn biết rằng ít nhất có một trong các đồng tiền là ngửa. Đâu là xác suất để cả hai đồng tiền đều là ngửa? Thông tin đã cho loại bỏ khả năng có hai mặt sấp. Ba sự kiện cơ bản còn lại hầu như bằng nhau, do đó ta suy ra mỗi sự kiện xảy ra với xác suất 1/3. Bởi chỉ một trong các sự kiện cơ bản này nêu hai mặt ngửa, nên đáp án cho câu hỏi của chúng ta là 1/3.

Xác suất có điều kiện hình thức hóa khái niệm biết trước một phần kết quả của một thí nghiệm. ***Xác suất có điều kiện*** của một sự kiện A

căn cứ vào một sự kiện B khác xảy ra sẽ được định nghĩa là

$$\Pr\{A \mid B\} = \frac{\Pr\{A \cap B\}}{\Pr\{B\}} \qquad (6.19)$$

mỗi khi Pr {B} $\neq$ 0. (Ta đọc "Pr $\{A \mid B\}$" là "xác suất của A căn cứ vào B.") Theo trực giác, do ta được biết sự kiện B xảy ra, nên sự kiện mà A cũng xảy ra là $A \cap B$. Nghĩa là, $A \cap B$ là tập hợp các kết quả ở đó cả A và B đều xảy ra. Do kết quả là một trong các sự kiện cơ bản trong B, ta chuẩn hóa [normalize] các xác suất của tất cả các sự kiện cơ bản trong B bằng cách chia chúng với Pr $\{B\}$, sao cho chúng tổng cộng là 1. Do đó, xác suất có điều kiện của A căn cứ vào B chính là tỷ số xác suất của sự kiện $A \cap B$ với xác suất của sự kiện B. Trong ví dụ trên đây, A là sự kiện mà cả hai đồng tiền đều ngửa, và B là sự kiện mà ít nhất một đồng tiền là mặt ngửa. Như vậy, Pr $\{A \mid B\} = (1/4)/(3/4) = 1/3$.

Hai sự kiện là ***độc lập*** nếu

$$\Pr\{A \cap B\} = \Pr\{A\}\Pr\{B\},$$

là tương đương, nếu Pr $\{B\} \neq 0$, theo điều kiện

$$\Pr\{A \mid B\} = \Pr\{A\}.$$

Ví dụ, giả sử hai đồng tiền cân được tung và các kết quả là độc lập. Vậy xác suất của hai mặt ngửa là (1/2) (1/2) = 1/4. Giờ đây giả sử một sự kiện là đồng tiền đầu tiên trúng các mặt ngửa và sự kiện kia là các đồng tiền trúng khác nhau. Mỗi sự kiện này xảy ra với xác suất 1/2, và xác suất mà cả hai sự kiện xảy ra là 1/4; như vậy, theo định nghĩa của sự độc lập, các sự kiện là độc lập—cho dù ta có thể cho rằng cả hai sự kiện tùy thuộc vào đồng tiền đầu tiên. Cuối cùng, giả sử các đồng tiền được hàn lại với nhau sao cho cả hai đều rơi các mặt ngửa hoặc đều rơi các mặt sấp và hai khả năng có thể bằng nhau. Vậy xác suất mà mỗi đồng tiền trúng các mặt ngửa là 1/2, nhưng xác suất mà cả hai đều trúng các mặt ngửa là $1/2 \neq (1/2)(1/2)$. Bởi vậy, sự kiện mà một đồng trúng các mặt ngửa và sự kiện đồng kia trúng các mặt ngửa sẽ không độc lập.

Một tập hợp $A_1, A_2, \ldots, A_n$ các sự kiện được gọi là ***độc lập theo từng cặp*** [pairwise independent] nếu

$$\Pr\{A_i \cap A_j\} = \Pr\{A_i\}\Pr\{A_j\}$$

với tất cả $1 \le i < j \le n$. Ta nói rằng chúng là ***độc lập*** (***tương hỗ***) nếu mọi tập hợp con $A_{i_1}, A_{i_2}, \ldots, A_{i_k}$ của tập hợp đó, ở đó $2 \le k \le n$ và $1 \le i_1 < i_2 < \cdots < i_k \le n$, thỏa

$$\Pr\{A_{i_1} \cap A_{i_2} \cap \cdots \cap A_{i_k}\} = \Pr\{A_{i_1}\}\Pr\{A_{i_2}\}\cdots\Pr\{A_{i_k}\}.$$

Ví dụ, giả sử ta tung hai đồng tiền cân. Cho A_1 là sự kiện mà đồng

tiền đầu tiên là các mặt ngửa, cho A_2 là sự kiện mà đồng tiền thứ hai là các mặt ngửa, và cho A_3 là sự kiện mà hai đồng tiền khác nhau. Ta có

$$\begin{aligned}
\Pr\{A_1\} &= 1/2\,, \\
\Pr\{A_2\} &= 1/2, \\
\Pr\{A_3\} &= 1/2, \\
\Pr\{A_1 \cap A_2\} &= 1/4\,, \\
\Pr\{A_1 \cap A_3\} &= 1/4\,, \\
\Pr\{A_2 \cap A_3\} &= 1/4\,, \\
\Pr\{A_1 \cap A_2 \cap A_3\} &= 0\,.
\end{aligned}$$

Bởi với $1 \le i < j \le 3$, ta có $\Pr\{A_i \cap A_j\} = \Pr\{A_i\}\Pr\{A_j\} = 1/4$, các sự kiện A_1, A_2, và A_3 độc lập theo từng cặp. Tuy nhiên, các sự kiện không độc lập tương hỗ, bởi vì $\Pr\{A_1\} \cap A_2 \cap A_3\} = 0$ và $\Pr\{A_1\}\Pr\{A_2\}\Pr\{A_3\} = 1/8 \ne 0$.

Định lý Bayes

Từ định nghĩa về xác suất có điều kiện (6.19), dẫn đến với hai sự kiện A và B, mỗi sự kiện có xác suất khác không,

$$\begin{aligned}
\Pr\{A \cap B\} &= \Pr\{B\}\Pr\{A \mid B\} \\
&= \Pr\{A\}\Pr\{B \mid A\}\,.
\end{aligned} \tag{6.20}$$

Giải với $\Pr\{A \mid B\}$, ta được

$$\Pr\{A \mid B\} = \frac{\Pr\{A\}\Pr\{B \mid A\}}{\Pr\{B\}}\,, \tag{6.21}$$

được xem là ***định lý Bayes***. Mẫu thức Pr {B} là một hằng chuẩn hóa mà ta có thể diễn tả lại như sau. Do $B = (B \cap A) \cup (B \cap A)$ và $B \cap A$ và $B \cap \overline{A}$ là các sự kiện loại trừ nhau,

$$\begin{aligned}
\Pr\{B\} &= \Pr\{B \cap A\} + \Pr\{B \cap \overline{A}\} \\
&= \Pr\{A\}\Pr\{B \mid A\} + \Pr\{\overline{A}\}\Pr\{B \mid \overline{A}\}.
\end{aligned}$$

Thay vào phương trình (6.21), ta được một dạng tương đương định lý Bayes:

$$\Pr\{A \mid B\} = \frac{\Pr\{A\}\Pr\{B \mid A\}}{\Pr\{A\}\Pr\{B \mid A\} + \Pr\{\overline{A}\}\Pr\{B \mid \overline{A}\}}$$

Định lý Bayes có thể giản lược sự tính toán các xác suất có điều kiện. Ví dụ, giả sử ta có một đồng tiền cân và một đồng tiền chênh luôn trúng các mặt ngửa. Ta làm một thí nghiệm bao gồm ba sự kiện độc lập: một trong hai đồng tiền được chọn ngẫu nhiên, đồng tiền được tung một lần, sau đó được tung lần nữa. Giả sử đồng tiền được chọn đã trúng các mặt ngửa cả hai lần. Đâu là xác suất mà nó bị chênh?

Ta giải bài toán này bằng định lý Bayes. Cho A là sự kiện mà đồng tiền chênh được chọn, và cho B là sự kiện mà đồng tiền trúng các mặt ngửa cả hai lần. Ta muốn xác định $\Pr\{A \mid B\}$. Ta có $\Pr\{A\} = 1/2$, $\Pr\{B \mid A\} = 1$, $\Pr\{\overline{A}\} = 1/2$, và $\Pr\{B \mid \overline{A}\} = 1/4$; do đó,

$$\Pr\{A \mid B\} = \frac{(1/2) \cdot 1}{(1/2) \cdot 1 + (1/2) \cdot (1/4)}$$

$$= 4/5 .$$

Bài tập

6.2-1

Chứng minh ***bất đẳng thức Bool***: Với bất kỳ dãy hữu hạn hoặc vô hạn khả đếm gồm các sự kiện $A_1, A_2, \ldots,$

$$\Pr\{A_1 \cup A_2 \cup \cdots\} \le \Pr\{A_1\} + \Pr\{A_2\} + \cdots \quad (6.22)$$

6.2-2

Giáo sư Rosencrantz tung một đồng tiền cân. Giáo sư Guildenstern tung hai đồng tiền cân. Đâu là xác suất mà Giáo sư Rosencrantz có được các mặt ngửa nhiều hơn Giáo sư Guildenstern?

6.2-3

Một cỗ bài có 10 lá, mỗi lá mang một số riêng biệt từ 1 đến 10, được xáo để trộn kỹ các lá bài. Ba lá bài lần lượt được gỡ bỏ ra khỏi cỗ bài từng lá một. Đâu là xác suất mà ba lá bài được lựa theo thứ tự đã sắp xếp (tăng)?

6.2-4 *

Bạn được giao một đồng tiền chênh, mà khi tung, nó cho ra một mặt ngửa với xác suất p (chưa biết), ở đó $0 < p < 1$. Nêu cách mô phỏng "tung đồng tiền" cân bằng cách xem xét nhiều lần tung. (*Mách nước.* Tung đồng tiền hai lần rồi kết xuất kết quả của lần tung đồng tiền cân đã mô phỏng hoặc lặp lại thí nghiệm.) Chứng minh đáp án là đúng.

6.2-5 *

Mô tả một thủ tục tiếp nhận dưới dạng đầu vào hai số nguyên a và b sao cho $0 < a < b$ và, dùng các lần tung đồng tiền cân, cho ra dưới dạng kết xuất các mặt ngửa với xác suất a/b và mặt sấp với xác suất $(b - a)/b$. Nêu một cận dựa trên số lượng dự trù các lần tung đồng tiền, sẽ là đa thức trong lg b.

6.2-6

Chứng minh rằng

$\Pr\{A \mid B\} + \Pr\{\overline{A} \mid B\} = 1.$

6.2-7

Chứng minh với bất kỳ tập hợp các sự kiện $A_1, A_2, \ldots, A_n$,

$$\Pr\{A_1 \cap A_2 \cap \ldots \cap A_n\} = \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \ldots \Pr\{A_n \mid A_1 \cap A_2 \cap \ldots \cap A_{n-1}\}.$$

6.2-8 *

Nêu cách kiến tạo một tập hợp *n* sự kiện độc lập thèo từng cặp [pairwise independent] nhưng sao cho bất kỳ tập hợp con $k > 2$ của chúng *không* độc lập tương hỗ.

6.2-9 *

Hai sự kiện *A* và *B* ***độc lập có điều kiện***, căn cứ vào *C*, nếu

$\Pr\{A \cap B \mid C\} = \Pr\{A \mid C\} \cdot \Pr\{B \mid C\}$.

Nêu một ví dụ đơn giản nhưng không tầm thường về hai sự kiện không độc lập song lại độc lập có điều kiện căn cứ vào một sự kiện thứ ba.

6.2-10 *

Bạn là một đấu thủ trong một cuộc thi đấu ở đó giải thưởng được giấu sau một trong ba bức màn. Bạn sẽ thắng giải nếu lựa đúng bức màn. Sau khi bạn chọn một bức màn nhưng trước khi bức màn được nâng lên, người chủ trì nâng một trong các bức màn kia, bày ra một bệ trống, và hỏi xem bạn có muốn chuyển từ bức màn đang được lựa sang bức màn còn lại hay không. Cơ may sẽ thay đổi ra sao nếu như bạn chuyển?

6.2-11 *

Một cai ngục đã chọn ngẫu nhiên một trong số ba tù nhân để thả tự do. Hai người kia sẽ bị xử tử. Lính gác biết người nào sẽ được thả tự do song bị cấm không được cung cấp cho bất kỳ tù nhân nào các thông tin liên quan tình trạng của họ. Ta hãy gọi tên các tù nhân là *X*, *Y*, và *Z*. Tù nhân *X* hỏi riêng lính gác ai trong số *Y* hoặc *Z* sẽ bị xử tử, biện luận rằng do anh ta đã biết sẵn ít nhất một trong số họ phải chết, lính gác sẽ không hé gì về tình trạng riêng của anh ta. Lính gác cho *X* biết rằng *Y* sẽ bị xử tử. Tù nhân *X* giờ đây cảm thấy mừng hơn, bởi anh ta hình dung rằng hoặc anh hoặc tù nhân *Z* sẽ được thả tự do, có nghĩa là xác suất thả tự do của anh ta giờ đây là 1/2. Anh ta có đúng không, hay cơ may của anh ta vẫn là 1/3? Giải thích.

6.3 Các biến ngẫu nhiên rời rạc

Một ***biến ngẫu nhiên (rời rạc)*** X là một hàm từ một không gian mẫu hữu hạn hoặc vô hạn khả đếm S đến các số thực. Nó kết hợp một số thực với từng kết quả khả dĩ của một thí nghiệm, cho phép ta làm việc với phép phân phối xác suất được cảm sinh trên tập hợp các con số kết quả. Các biến ngẫu nhiên cũng có thể được định nghĩa với các không gian mẫu vô hạn bất khả đếm, nhưng chúng sẽ làm nảy sinh các vấn đề về kỹ thuật không cần thiết để cập theo các mục tiêu của chúng ta. Từ đây, ta mặc nhận rằng các biến ngẫu nhiên là rời rạc.

Với một biến ngẫu nhiên X và một số thực x, ta định nghĩa sự kiện $X = x$ là $\{s \in S : X(s) = x\}$; như vậy,

$$\Pr\{X = x\} = \sum_{\{s \in S: X(s)=x\}} \Pr\{s\}.$$

Hàm

$$f(x) = Pr\{X = x\}$$

là ***hàm mật độ xác suất*** của biến ngẫu nhiên X. Từ các tiên đề xác suất, $\Pr\{X = x\} \geq 0$ và $\sum_x \Pr\{X = x\} = 1$.

Để lấy ví dụ, ta xét thí nghiệm lắc một cặp xúc xắc bình thường, 6 mặt. Có 36 sự kiện cơ bản khả dĩ trong không gian mẫu. Ta mặc nhận rằng phép phân phối xác suất là đều [uniform], sao cho mỗi sự kiện cơ bản $s \in S$ có khả năng như nhau: $\Pr\{s\} = 1/36$. Định nghĩa biến ngẫu nhiên X là cực đại của hai giá trị nêu trên xúc xắc. Ta có $\Pr\{X = 3\} = 5/36$, bởi X gán một giá trị 3 cho 5 của 36 sự kiện cơ bản khả dĩ, tức là, (1, 3), (2, 3), (3, 3), (3, 2), và (3, 1).

Ta thường định nghĩa vài biến ngẫu nhiên trên cùng không gian mẫu. Nếu X và Y là các biến ngẫu nhiên, hàm

$$f(x, y) = \Pr\{X = x \text{ và } Y = y\}$$

là ***hàm mật độ xác suất nối*** [joint probability density function] của X và Y. Với một giá trị cố định y,

$$\Pr\{Y = y\} = \sum_x \Pr\{X = x \text{ và } Y = y\},$$

và cũng vậy, với một giá trị cố định x,

$$\Pr\{X = x\} = \sum_y \Pr\{X = x \text{ và } Y = y\}.$$

Dùng định nghĩa (6.19) về xác suất có điều kiện, ta có

$$\Pr\{X = x \mid Y = y\} = \frac{\Pr\{X = x \text{ và } Y = y\}}{\Pr\{Y = y\}}$$

Ta định nghĩa hai biến ngẫu nhiên X và Y là ***độc lập*** nếu với tất cả x và y, các sự kiện $X = x$ và $Y = y$ là độc lập hoặc, tương đương, nếu với tất cả x và y, ta có $\Pr\{X = x \text{ và } Y = y\} = \Pr\{X = x\}\Pr\{Y = y\}$.

Cho một tập hợp các biến ngẫu nhiên được định nghĩa trên cùng không gian mẫu, ta có thể định nghĩa các biến ngẫu nhiên mới dưới dạng các tổng, các tích, hoặc các hàm khác của các biến ban đầu.

Giá trị dự trù của một biến ngẫu nhiên

Tóm tắt đơn giản và hữu ích nhất về phép phân phối một biến ngẫu nhiên chính là "trung bình"của các giá trị nó nhận. ***Giá trị dự trù*** [expected value] (hoặc, đồng nghĩa, ***giá trị kỳ vọng*** [expectation] hay ***giá trị trung bình*** [mean]) của một biến ngẫu nhiên rời rạc X là

$$\mathrm{E}[X] = \sum_x x \Pr\{X = x\}, \tag{6.23}$$

được định nghĩa kỹ nếu tổng là hữu hạn hoặc hội tụ tuyệt đối. Đôi lúc giá trị kỳ vọng của X được ký hiệu là μ_X hoặc, khi biến ngẫu nhiên đã rõ theo ngữ cảnh, đơn giản là μ.

Xem xét một trò chơi ở đó bạn tung hai đồng tiền cân. Bạn ăn \$3 với mỗi mặt ngửa nhưng thua \$2 với mỗi mặt sấp. Giá trị dự trù của biến ngẫu nhiên X biểu thị phần ân của bạn là

$$\begin{aligned}\mathrm{E}[X] &= 6 \cdot \Pr\{2\text{ H'S}\} + 1 \cdot \Pr\{1\text{ H}, 1\text{ T}\} - 4 \cdot \Pr\{2\text{ T'S}\} \\ &= 6(1/4) + 1(1/2) - 4(1/4) \\ &= 1 .\end{aligned}$$

Giá trị kỳ vọng của tổng hai biến ngẫu nhiên là tổng các giá trị kỳ vọng của chúng, nghĩa là,

$$\mathrm{E}[X + Y] = \mathrm{E}[X] + \mathrm{E}[Y], \tag{6.24}$$

mỗi khi $\mathrm{E}[X]$ và $\mathrm{E}[Y]$ được định nghĩa. Tính chất này khai triển các phép lấy tổng hữu hạn và hội tụ tuyệt đối của các giá trị kỳ vọng.

Nếu X là một biến ngẫu nhiên bất kỳ, mọi hàm $g(x)$ sẽ định nghĩa một biến ngẫu nhiên $g(X)$ mới. Nếu giá trị kỳ vọng của $g(X)$ được định nghĩa, thì

$$\mathrm{E}[g(X)] = \sum_x g(x) \Pr\{X = x\} .$$

Cho $g(x) = ax$, ta có với bất kỳ hằng a,

$$\mathrm{E}[aX] = a\mathrm{E}[X] . \tag{6.25}$$

Bởi vậy, các giá trị kỳ vọng là tuyến tính: với bất kỳ hai biến ngẫu

nhiên X và Y và bất kỳ hằng a,

$$E[aX + Y] = aE[X] + E[Y]. \tag{6.26}$$

Khi hai biến ngẫu nhiên X và Y độc lập và từng biến có một giá trị kỳ vọng đã định nghĩa,

$$\begin{aligned} E[XY] &= \sum_x \sum_y xy \Pr\{X = x \text{ và } Y = y\} \\ &= \sum_x \sum_y xy\Pr\{X = x\} \Pr\{Y = y\} \\ &= \left(\sum_x x\Pr\{X = x\}\right) \left(\sum_y y \Pr\{Y = y\}\right) \\ &= E[X]E[Y]. \end{aligned}$$

Nói chung, khi n biến ngẫu nhiên $X_1, X_2, \ldots, X_n$ độc lập tương hỗ,

$$E[X_1 X_2 \ldots X_n] = E[X_1]E[X_2]\ldots E[X_n] \tag{6.27}$$

Khi một biến ngẫu nhiên X nhận các giá trị từ các số tự nhiên $\mathbf{N} = \{0, 1, 2, \ldots\}$, ta có một công thức thích hợp cho giá trị kỳ vọng của nó:

$$\begin{aligned} E[X] &= \sum_{i=0}^{\infty} i\Pr\{X = i\} \\ &= \sum_{i=0}^{\infty} i(\Pr\{X \geq i\} - \Pr\{X \geq i+1\}) \\ &= \sum_{i=1}^{\infty} \Pr\{X \geq i\}, \end{aligned} \tag{6.28}$$

bởi mỗi số hạng $\Pr\{X \geq i\}$ được cộng i lần và trừ đi $i - 1$ lần (ngoại trừ $\Pr\{X \geq 0\}$, được cộng 0 lần và không được trừ gì cả).

Phương sai và độ lệch chuẩn

Phương sai [variance] của một biến ngẫu nhiên X có giá trị trung bình $E[X]$ là

$$\begin{aligned} \text{Var}[X] &= E[(X - E[X])^2] \\ &= E[X^2 - 2XE[X] + E^2[X]] \\ &= E[X^2 - 2E[XE[X]] + E^2[X] \\ &= E[X^2] - 2E^2[X] + E^2[X] \\ &= E[X^2] - E^2[X]. \end{aligned} \tag{6.29}$$

Biện luận cho các đẳng thức $E[E^2[X]] = E^2[X]$ và $E[XE[X]] = E^2[X]$ đó là $E[X]$ không phải là một biến ngẫu nhiên nhưng đơn giản là một số thực, có nghĩa là áp dụng phương trình (6.25) (với $a = E[X]$). Phương trình (6.29) có thể được viết lại để có một biểu thức với giá trị kỳ vọng của bình phương của một biến ngẫu nhiên:

$$E[X^2] = \text{Var}[X] + E^2[X]. \tag{6.30}$$

Phương sai của một biến ngẫu nhiên X và phương sai của aX có liên đới:

$\mathrm{Var}[aX] = a^2 \mathrm{Var}[X]$.

Khi X và Y là các biến ngẫu nhiên độc lập,

$\mathrm{Var}[X + Y] = \mathrm{Var}[X] + \mathrm{Var}[Y]$.

Nói chung, nếu n biến ngẫu nhiên $X_1, X_2, \ldots, X_n$ độc lập theo từng cặp, thì

$$\mathrm{Var}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathrm{Var}[X_i]. \tag{6.31}$$

Độ lệch chuẩn [standard deviation] của một biến ngẫu nhiên X là căn bậc hai dương của phương sai của X. Độ lệch chuẩn của một biến ngẫu nhiên X đôi lúc được ký hiệu là σ_X hoặc đơn giản là σ khi biến ngẫu nhiên X đã rõ theo ngữ cảnh. Với hệ ký hiệu này, phương sai của X được ký hiệu là σ^2

Bài tập

6.3-1

Lăn hai con xúc xắc bình thường, 6 cạnh. Đâu là giá trị kỳ vọng của tổng hai giá trị đang hiện? Đâu là giá trị kỳ vọng của cực đại của hai giá trị đang hiện?

6.3-2

Một mảng $A[1..n]$ chứa n con số riêng biệt được sắp xếp theo ngẫu nhiên, với mỗi phép hoán vị n con số có khả năng như nhau. Đâu là giá trị kỳ vọng của chỉ số của thành phần cực đại trong mảng? Đâu là giá trị kỳ vọng của chỉ số thành phần cực tiểu trong mảng?

6.3-3

Một trò chơi lễ hội bao gồm ba con xúc xắc trong một chiếc lồng. Một người chơi có thể cá một đôla trên bất kỳ trong các số 1 đến 6. Chiếc lồng được lắc, và việc được thua như sau. Nếu con số của người chơi không xuất hiện trên bất kỳ con xúc xắc nào, anh ta thua tiền cược của mình. Bằng không, nếu số của anh ta xuất hiện trên chính xác k của ba con xúc xắc, với $k = 1, 2, 3$, anh ta giữ lại tiền của mình và thắng thêm k đôla. Đâu là phần thu được dự trù từ việc chơi một lần?

6.3-4 *

Cho X và Y là các biến ngẫu nhiên độc lập. Chứng minh $f(X)$ và $g(Y)$ độc lập với bất kỳ chọn lựa nào của các hàm f và g.

6.3-5 *

Cho X là một biến ngẫu nhiên không âm, và giả sử rằng E [X] được định nghĩa kỹ. Chứng minh ***bất đẳng thức Markov:***

$$\Pr\{X \geq t\} \leq E[X]/t \qquad (6.32)$$

for all $t > 0$.

6.3-6 *

Cho S là một không gian mẫu, và cho X và X' là các biến ngẫu nhiên sao cho $X(s) \geq X'(s)$ với tất cả $s \in S$. Chứng minh rằng với bất kỳ hằng thực t,

$$\Pr\{X \geq t\} \geq \Pr\{X' \geq t\}.$$

6.3-7

Cái nào lớn hơn: giá trị kỳ vọng của bình phương của một biến ngẫu nhiên, hay bình phương của giá trị kỳ vọng của nó?

6.3-8

Chứng tỏ với bất kỳ biến ngẫu nhiên X chỉ nhận các giá trị 0 và 1, ta có $\mathrm{Var}[X] = E[X]\,E[1 - X]$.

6.3-9

Chứng minh $\mathrm{Var}[aX] = a^2\mathrm{Var}[x]$ từ định nghĩa (6.29) về phương sai.

6.4 Các phép phân phối nhị thức và theo cấp số nhân

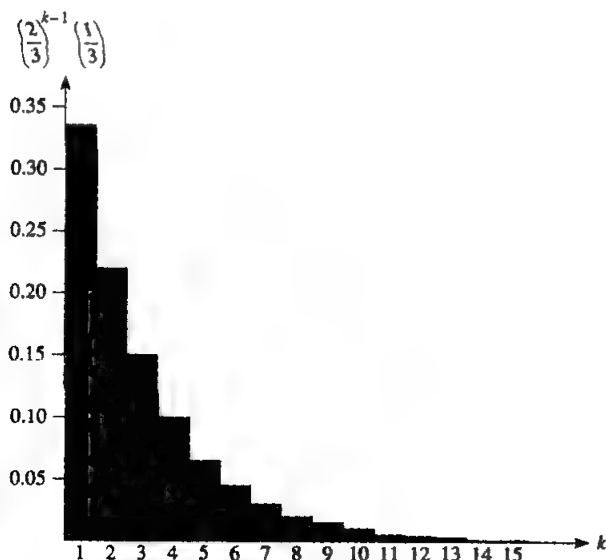
Một lần tung đồng tiền là một minh dụ [instance] của một ***lần thử Bernoulli*** [Bernoulli trial], được định nghĩa dưới dạng một thí nghiệm chỉ có hai kết quả khả dĩ: ***lần thành công***, xảy ra với xác suất p, và ***lần thất bại***, xảy ra với xác suất $q = 1 - p$. Khi nói chung là ***các lần thử Bernoulli***, ta ám chỉ các lần thử độc lập tương hỗ và (trừ phi ta nói cụ thể khác đi), mỗi lần thử có cùng xác suất p lần thành công. Hai phép phân phối quan trọng nảy sinh từ các lần thử Bernoulli đó là: phép phân phối cấp số nhân và phép phân phối nhị thức.

Phép phân phối cấp số nhân

Giả sử ta có một dãy các lần thử Bernoulli, mỗi lần thử có một xác suất p lần thành công và một xác suất $q = 1 - p$ lần thất bại. Bao nhiêu lần thử xảy ra trước khi ta được một lần thành công? Cho biến ngẫu nhiên X là số lần thử cần có để được một lần thành công. Thì X có các giá trị trong miền giá trị $\{1, 2, \ldots\}$, và với $k \geq 1$,

$$\Pr\{X = k\} = q^{k-1}p\,, \tag{6.33}$$

bởi ta có $k - 1$ lần thất bại trước một lần thành công. Một phép phân phối xác suất thỏa phương trình (6.33) được gọi là ***phép phân phối cấp số nhân*** [geometric distribution]. Hình 6.1 minh họa một phép phân phối như vậy.



Hình 6.1 Một phép phân phối cấp số nhân có xác suất $p = 1/3$ lần thành công và xác suất $q = 1 - p$ lần thất bại. Giá trị kỳ vọng của phép phân phối là $1/p = 3$.

Giả sử $p < 1$, giá trị kỳ vọng của một phép phân phối cấp số nhân có thể được tính bằng đồng nhất thức (3.6):

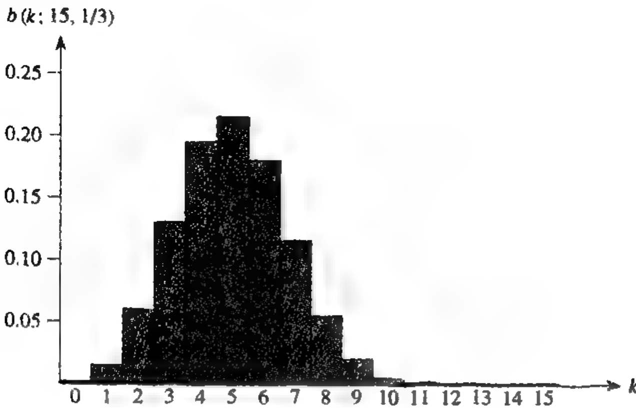
$$\begin{aligned} \mathrm{E}[X] &= \sum_{k=1}^{\infty} kq^{k-1}p \\ &= \sum_{k=0}^{\infty} kq^{k} \\ &= \frac{p}{q} \cdot \frac{q}{(1-q)^2} \\ &= 1/p\,. \end{aligned} \tag{6.34}$$

Như vậy, tính bình quân, nó mất $1/p$ lần thử trước khi ta được một lần thành công, một kết quả trực giác. Phương sai, có thể được tính tương tự, là

$\mathrm{Var}[X] = q/p^2$. (6.35)

Để lấy ví dụ, giả sử ta lăn liên tục hai con xúc xắc cho đến khi được một số bảy hoặc số mười một. Từ 36 kết quả khả dĩ, 6 cho một số bảy và 2 cho ra một số mười một. Như vậy, xác suất lần thành công là $p = 8/$

36 = 2/9, và ta phải lăn trung bình $1/p = 9/2 = 4.5$ lần để được một số bảy hoặc mười một.



Hình 6.2 Phép phân phối nhị thức $b(k;\ 15,\ 1/3)$ kết quả của $n = 15$ lần thử Bernoulli, mỗi lần với xác suất $p = 1/3$ lần thành công. Giá trị kỳ vọng của phép phân phối là $np = 5$.

Phép phân phối nhị thức

Bao nhiêu lần thành công xảy ra trong n lần thử Bernoulli, ở đó một lần thành công xảy ra với xác suất p và một lần thất bại với xác suất $q = 1 - p$? Định nghĩa biến ngẫu nhiên X là số lượng lần thành công trong n lần thử. Thì X có các giá trị trong miền giá trị $\{0,1,\ldots, n\}$, và với $k = 0,\ldots, n$,

$$\Pr\{X = k\} = \binom{n}{k} p^k q^{n-k}, \tag{6.36}$$

bởi có $\binom{n}{k}$ cách để chọn k nào của n lần thử là lần thành công, và xác suất mà mỗi lần thành công xảy ra là $p^k q^{n-k}$. Một phép phân phối xác suất thỏa phương trình (6.36) được gọi là một ***phép phân phối nhị thức*** [binomial distribution]. Để tiện dụng, ta định nghĩa họ phép phân phối nhị thức bằng hệ ký hiệu

$$b(k; n, p) = \binom{n}{k} p^k (1-p)^{n-k}. \tag{6.37}$$

Hình 6.2 minh họa một phép phân phối nhị thức. Tên "nhị thức" xuất xứ từ sự việc (6.37) là số hạng thứ k của khai triển $(p + q)^n$. Bởi vậy, do $p + q = 1$,

$$\sum_{k=0}^{\infty} b(k; n, p) = 1, \tag{6.38}$$

như được quy định bởi tiên đề 2 của các tiên đề xác suất.

Ta có thể tính toán giá trị kỳ vọng của một biến ngẫu nhiên có một phép phân phối nhị thức từ các phương trình (6.14) và (6.38). Cho X là một biến ngẫu nhiên theo phép phân phối nhị thức $b(k; n, p)$, và cho $q = 1 - p$. Theo định nghĩa về giá trị kỳ vọng, ta có

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{k=0}^{n} k b(k; n, p) \\
&= \sum_{k=1}^{n} k \binom{n}{k} p^k q^{n-k} \\
&= np \sum_{k=1}^{n} \binom{n-1}{k-1} p^k q^{(n-1)-k} \\
&= np \sum_{k=0}^{n-1} \binom{n-1}{k} p^k q^{(n-1)-k} \\
&= np \sum_{k=0}^{n-1} b(k; n-1, p) \\
&= np . \qquad (6.39)
\end{aligned}$$

Nhờ dùng tính chất tuyến tính của giá trị kỳ vọng, ta có thể có được cùng kết quả mà ít đại số học hơn đáng kể. Cho X_i là biến ngẫu nhiên mô tả số lượng lần thành công trong lần thử thứ i. Thì $\mathrm{E}[X_i] = p \bullet 1 + q \bullet 0 = p$, và bằng tính chất tuyến tính của giá trị kỳ vọng (6.26), số lượng lần thành công dự trù với n lần thử là

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] \\
&= \sum_{i=1}^{n} \mathrm{E}[X_i] \\
&= \sum_{i=1}^{n} p \\
&= np .
\end{aligned}$$

Có thể dùng cùng cách tiếp cận có thể để tính toán phương sai của phép phân phối. Dùng phương trình (6.29), ta có $\mathrm{Var}[X_i] = \mathrm{E}[X_i^2] - \mathrm{E}^2[X_i]$. Bởi X_i chỉ nhận các giá trị 0 và 1, nên ta có $\mathrm{E}[X_i^2] = \mathrm{E}[X_i] = p$, và do đó

$$\mathrm{Var}[X_i] = p - p^2 = pq . \qquad (6.40)$$

Để tính toán phương sai của X, ta vật dụng sự độc lập của n lần thử; như vậy, bằng phương trình (6.31),

$$\begin{aligned}
[X] &= \mathrm{Var}\left[\sum_{i=1}^{n} X_i\right] \\
&= \sum_{i=1}^{n} \mathrm{Var}[X_i] \\
&= \sum_{i=1}^{n} pq \\
&= npq . \qquad (6.41)
\end{aligned}$$

Như đã thấy trong Hình 6.2, phép phân phối nhị thức $b(k; n, p)$ gia tăng khi k chạy từ 0 đến n cho đến khi nó đụng giá trị trung bình np, rồi nó giảm. Có thể chứng minh rằng phép phân phối luôn ứng xử theo cách này bằng cách xem xét tỷ số của các số hạng kế tiếp:

$$\begin{aligned}\frac{b(k; n, p)}{b(k-1; n, p)} &= \frac{\binom{n}{k} p^k q^{n-k}}{\binom{n}{k-1} p^{k-1} q^{n-k+1}} \\ &= \frac{n!(k-1)!(n-k+1)!p}{k!(n-k)!n!q} \\ &= \frac{(n-k+1)p}{kq} \\ &= 1 + \frac{(n+1)p-k}{kq} .\end{aligned} \tag{6.42}$$

Tỷ số này chính xác lớn hơn 1 khi $(n + 1)p - k$ là dương. Bởi vậy, $b(k; n, p) > b(k - 1; n, p)$ với $k < (n + 1)p$ (phép phân phối tăng), và $b(k; n, p) < b(k - 1; n, p)$ với $k > (n + 1)p$ (phép phân phối giảm). Nếu $k = (n + 1)p$ là một số nguyên, thì $b(k; n, p) = b(k - 1; n, p)$, như vậy phép phân phối có hai cực đại [maxima]: tại $k = (n + 1)p$ và tại $k - 1 = (n + 1)p - 1 = np - q$. Bằng không, nó đạt đến một cực đại tại số nguyên duy nhất k nằm trong miền giá trị $np - q < k < (n + 1)p$.

Bổ đề sau đây cung cấp một cận trên dựa trên phép phân phối nhị thức.

Bổ đề 6.1

Cho $n \geq 0$, cho $0 < p < 1$, cho $q = 1 - p$, và cho $0 \leq k \leq n$. Thì

$$b(k; n, p) \leq \left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k} .$$

Chứng minh

Dùng phương trình (6.10), ta có

$$\begin{aligned}b(k; n, p) &= \binom{n}{k} p^k q^{n-k} \\ &\leq \left(\frac{n}{k}\right)^k \left(\frac{n}{n-k}\right)^{n-k} p^k q^{n-k} \\ &= \left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k}\end{aligned}$$

Bài tập

6.4-1

Xác minh tiên đề 2 của các tiên đề xác suất với phép phân phối cấp số nhân.

6.4-2

Trung bình ta phải tung bao nhiêu lần 6 đồng tiền cân trước khi được 3 mặt ngửa và 3 mặt sấp?

6.4-3

Chứng tỏ $b(k; n, p) = b(n - k; n, q)$, ở đó $q = 1 - p$.

6.4-4

Chứng tỏ giá trị cực đại [maximum] của phép phân phối nhị thức $b(k; n, p)$ là xấp xỉ $1/\sqrt{2\pi npq}$, ở đó $q = 1 - p$.

6.4-5 *

Chứng tỏ xác suất của không lần thành công trong n lần thử Bernoulli, mỗi lần thử với xác suất $p = 1 / n$, xấp xỉ là $1/e$. Chứng tỏ xác suất của chính xác một lần thành công cũng xấp xỉ là $1/e$.

6.4-6 *

Giáo sư Rosencrantz tung một đồng tiền cân n lần, và Giáo sư Guildenstern cũng vậy. Chứng tỏ xác suất mà họ có cùng số lượng mặt ngửa là $\binom{2n}{n} / 4^n$. (*Mách nước:* Với Giáo sư Rosencrantz, ta gọi một mặt ngửa là một lần thành công; với Giáo sư Guildenstern, ta gọi một mặt sấp là một lần thành công.) Dùng biện luận của bạn để xác minh đồng nhất thức

$$\sum_{k=0}^{n} \binom{n}{k}^2 = \binom{2n}{n}.$$

6.4-7 *

Chứng tỏ với $0 \le k \le n$,

$$b(k; n, 1/2) \le 2^{nH(k/n)-n},$$

ở đó $H(x)$ là hàm entropi (6.13).

6.4-8 *

Xem xét n lần thử Bernoulli, ở đó với $i = 1, 2, \ldots, n$, lần thử thứ i có xác suất p_i thành công, và cho X là biến ngẫu nhiên thể hiện tổng số lần thành công. Cho $p \ge p_i$ với tất cả $i = 1, 2, \ldots, n$. Chứng minh rằng với $1 \le k \le n$,

$$\Pr\{X < k\} \le \sum_{i=0}^{k-1} b(i; n, p)$$

6.4-9 *

Cho X là biến ngẫu nhiên với tổng số lần thành công trong một tập hợp A gồm n lần thử Bernoulli, ở đó lần thử thứ i có xác suất p_i thành công, và cho X' là biến ngẫu nhiên với tổng số lần thành công trong một tập hợp thứ hai A' gồm n lần thử Bernoulli, ở đó lần thử thứ i có một xác suất $p'_i \ge p_i$; thành công. Chứng minh rằng với $0 \le k \le n$,

$$\Pr\{X' \ge k\} \ge \Pr\{X \ge k\} .$$

(*Mách nước:* Nêu cách có được các lần thử Bernoulli trong A' bằng một thí nghiệm liên quan đến các lần thử của A, và dùng kết quả của Bài tập 6.3-6.)

---

* 6.5 Các mặt sấp của phép phân phối nhị thức

Xác suất có ít nhất, hoặc tối đa, k thành công trong n lần thử Bernoulli, mỗi lần với xác suất p thành công, thường đáng để quan tâm hơn so với xác suất có chính xác k thành công. Trong đoạn này, ta nghiên cứu các ***mặt sấp*** của phép phân phối nhị thức: hai vùng của phép phân phối $b(k; n, p)$ cách xa với giá trị trung bình np. Ta sẽ chứng minh vài cận quan trọng trên (tổng của tất cả số hạng trong) một mặt sấp.

Trước tiên, ta cung cấp một cận trên mặt sấp phải của phép phân phối $b(k; n, p)$. Các cận trên mặt sấp trái có thể được xác định bằng cách đảo các vai trò [roles] của các lần thành công và các lần thất bại.

Định lý 6.2

Xem xét một dãy n lần thử Bernoulli, ở đó lần thành công xảy ra với xác suất p. Cho X là biến ngẫu nhiên thể hiện tổng số lần thành công. Thì với $0 \le k \le n$, xác suất của ít nhất k thành công là

$$\Pr\{X \ge k\} = \sum_{i=k}^{n} b(i; n, p) \le \binom{n}{k} p^k .$$

Chứng minh

Ta vật dụng bất đẳng thức (6.15)

$$\binom{n}{k+i} \le \binom{n}{k}\binom{n-k}{i} .$$

Ta có

$$
\begin{aligned}
\Pr\{X \geq k\} &= \sum_{i=k}^{n} \\
&= \sum_{i=0}^{n-k} b(k+i, n, p) \\
&= \sum_{i=0}^{n-k} \binom{n}{k+i} p^{k+i}(1-p)^{n-(k+i)} \\
&\leq \sum_{i=0}^{n-k} \binom{n}{k}\binom{n-k}{i} p^{k+i}(1-p)^{n-(k+i)} \\
&= \binom{n}{k} p^k \sum_{i=0}^{n-k} \binom{n-k}{i} p^i(1-p)^{(n-k)-i} \\
&= \binom{n}{k} p^k \sum_{i=0}^{n-k} b(i; n-k, p) \\
&= \binom{n}{k} p^k
\end{aligned}
$$

bởi $\sum_{i=0}^{n-k} b(i; n-k, p) = 1$ theo phương trình (6.38).

Hệ luận dưới đây phát biểu lại định lý với mặt sấp trái của phép phân phối nhị thức. Nói chung, chúng tôi để nó cho bạn đọc thích ứng các cận từ mặt sấp này sang mặt sấp kia.

Hệ luận 6.3

Xem xét một dãy n lần thử Bernoulli, ở đó lần thành công xảy ra với xác suất p. Nếu X là biến ngẫu nhiên thể hiện tổng số lần thành công, thì với $0 \leq k \leq n$, xác suất của tối đa k thành công là

$$
\begin{aligned}
\Pr\{X \leq k\} &= \sum_{i=0}^{k-1} b(i; n, p) \\
&\leq \binom{n}{n-k}(1-p)^{n-k} \\
&= \binom{n}{k}(1-p)^{n-k} .
\end{aligned}
$$

Cận kế tiếp của chúng ta tập trung vào mặt sấp trái của phép phân phối nhị thức. Cách xa với Far từ giá trị trung bình, số lượng các lần thành công trong mặt sấp trái giảm bớt theo hàm mũ, như định lý dưới đây nêu rõ.

Định lý 6.4

Xem xét một dãy n lần thử Bernoulli, ở đó lần thành công xảy ra với xác suất p và lần thất bại với xác suất $q = 1 - p$. Cho X là biến ngẫu nhiên thể hiện tổng số lần thành công. Thì với $0 < k < np$, xác suất của ít hơn k thành công là

$$\Pr\{X < k\} = \sum_{i=0}^{k-1} b(i; n, p) < \frac{kq}{np - k} b(k; n, p) .$$

Chứng minh

Ta định cận chuỗi $\sum_{i=0}^{n-k} b(i; n, p)$ bằng một chuỗi cấp số nhân dùng kỹ thuật trong Đoạn 3.2. Với $i = 1, 2, \ldots, k$, ta có từ phương trình (6.42),

$$\frac{b(i - 1); n, p)}{b(i; n, p)} = \frac{iq}{(n - i + 1)p} < \left(\frac{i}{n - i}\right)\left(\frac{q}{p}\right) \leq \left(\frac{k}{n - k}\right)\left(\frac{q}{p}\right) .$$

Nếu ta cho

$$x = \left(\frac{k}{n - k}\right)\left(\frac{q}{p}\right) < 1 .$$

thì dẫn đến

$b(i - 1; n, p) < x\, b(i; n, p)$ với $0 < i \leq k$. Lặp lại, ta được

$b(i; n, p) < x^{k-i} b(k; n, p)$

với $0 \leq i < k$, và do đó

$$\begin{aligned} \sum_{i=0}^{k-1} b(i; n, p) &< \sum_{i=0}^{k-1} x^{k-i} b(k; n, p) \\ &< b(k; n, p) \sum_{i=1}^{\infty} x^i \\ &= \frac{x}{1 - x} b(k; n, p) \\ &= \frac{kq}{np - k} b(k; n, p) . \end{aligned}$$

Khi $k \leq np/2$, ta có $kq/(np - k) \leq 1$, có nghĩa là $b(k; n, p)$ định cận [bounds] tổng của tất cả các số hạng nhỏ hơn k. Để lấy ví dụ, giả sử ta tung n đồng tiền cân. Dùng $p = 1/2$ và $k = n/4$, Định lý 6.4 cho ta biết xác suất có được ít hơn $n/4$ mặt ngửa là nhỏ hơn xác suất có được chính xác $n/4$ mặt ngửa. Hơn nữa, với bất kỳ $r \geq 4$, xác suất có được ít hơn n/r mặt ngửa là nhỏ hơn có được chính xác n/r mặt ngửa. Định lý 6.4 cũng có thể khá hữu ích khi kết hợp với các cận trên dựa trên phép phân phối nhị thức, như Bổ đề 6.1.

Có thể dùng phương pháp tương tự để xác định một cận dựa trên mặt sấp phải.

Hệ luận 6.5

Xem xét một dãy n lần thử Bernoulli, ở đó lần thành công xảy ra với xác suất p. Cho X là biến ngẫu nhiên thể hiện tổng số lần thành công. Thì với $np < k < n$, xác suất của trên k thành công là

$$\begin{aligned} \Pr\{X > k\} &= \sum_{i=k+1}^{n} b(k; n, p) \\ &< \frac{(n-k)p}{k-np} b(k; n, p) . \end{aligned}$$

Định lý kế tiếp xem xét n lần thử Bernoulli, mỗi lần với một xác suất p_i thành công, với $i = 1, 2,..., n$. Như hệ luận tiếp theo nêu rõ, ta có thể dùng định lý để cung cấp một cận trên mặt sấp phải của phép phân phối nhị thức bằng cách ấn định $p_i = p$ với mỗi lần thử.

Định lý 6.6

Xem xét một dãy n lần thử Bernoulli, ở đó trong lần thử thứ i, với $i =$ 1, 2,..., n, lần thành công xảy ra với xác suất p_i và lần thất bại xảy ra với xác suất $q_i = 1 - p_i$. Cho X là biến ngẫu nhiên mô tả tổng số lần thành công, và cho $\mu = E[X]$. Thì với $r > \mu$,

$$\Pr\{X - \mu \geq r\} \leq \left(\frac{\mu e}{r}\right)^r .$$

Chứng minh

Bởi với bất kỳ $\alpha > 0$, hàm eax tăng ngặt trong x,

$$\Pr\{X - \mu \geq r\} = \Pr\{e^{\alpha(X-\mu)} \geq e^{\alpha r}\} ,$$

ở đó α sẽ được xác định sau. Dùng bất đẳng thức Markov (6.32), ta được

$$\Pr\{X - \mu \geq r\} \leq E[e^{\alpha(X-\mu)}] e^{-\alpha r} . \qquad (6.43)$$

Phần chính của phép chứng minh bao gồm việc định cận E $[e^{\alpha(X-\mu)}]$ và thay một giá trị thích hợp cho a trong bất đẳng thức (6.43). Trước tiên, ta đánh giá E $(e^{\alpha(X-\mu)}]$. Với $i = 1, 2,..., n$, cho X_i là biến ngẫu nhiên bằng 1 nếu lần thử Bernoulli thứ i là một lần thành công và 0 nếu nó là một lần thất bại. Như vậy,

$$X = \sum_{i=1}^{n} X_i$$

và

$$X - \mu = \sum_{i=1}^{n} (X_i - p_i) .$$

Thay cho $X - \mu$, ta được

$$\begin{aligned} E[e^{\alpha(X-\mu)}] &= E\left[\prod_{i=1}^{n} e^{\alpha(X_i-p_i)}\right] \\ &= \prod_{i=1}^{n} E[e^{\alpha(X_i-p_i)}] , \end{aligned}$$

theo như (6.27), bởi sự độc lập tương hỗ của các biến ngẫu nhiên X_i hàm ý sự độc lập tương hỗ của các biến ngẫu nhiên $e^{\alpha}(X_i - p_i)$ (xem Bài tập 6.3-4). Theo định nghĩa về giá trị kỳ vọng,

$$\begin{aligned} E[e^{\alpha(X_i-p_i)}] &= e^{\alpha(1-p_i)} p_i + e^{\alpha(0-p_i)} q_i \\ &= p_i e^{\alpha q_i} + q_i e^{-\alpha p_i} \\ &\le p_i e^{\alpha} + 1 \qquad (6.44) \\ &\le exp(p_i e^{\alpha}) , \end{aligned}$$

ở đó $exp(x)$ thể hiện hàm mũ: $\exp(x) = e^x$. (Bất đẳng thức (6.44) là do các bất đẳng thức $\alpha > 0$, $q \le 1$, $e^{\alpha q} \le e^{\alpha}$, và $e^{-\alpha p} \le 1$, và dòng cuối là do bất đẳng thức (2.7)). Bởi vậy,

$$\begin{aligned} E[e^{\alpha(X-\mu)}] &\le \prod_{i=1}^{n} \exp(p_i e^{\alpha}) \\ &= \exp(\mu e^{\alpha}) , \end{aligned}$$

bởi $\mu = \sum_{i=1}^{n} p_i$. Do đó, từ bất đẳng thức (6.43), dẫn đến

$$\Pr\{X - \mu \ge r\} \le \exp(\mu e^{\alpha} - \alpha r) . \qquad (6.45)$$

Chọn $\alpha = \ln(r/\mu)$ (xem Bài tập 6.5-6), ta được

$$\begin{aligned} \Pr\{X - \mu \ge r\} &\le \exp(\mu e^{\ln(r/\mu)} - r\ln(r/\mu)) \\ &= \exp(r - r\ln(r/\mu)) \\ &= \frac{e^r}{(r/\mu)^r} \\ &= \left(\frac{\mu e}{r}\right)^r . \end{aligned}$$

Khi được áp dụng cho các lần thử Bernoulli ở đó mỗi lần thử có cùng xác suất thành công, Định lý 6.6 cho ra hệ luận dưới đây định cận mặt sấp phải của một phép phân phối nhị thức.

Hệ luận 6.7

Xem xét một dãy n lần thử Bernoulli, ở đó trong mỗi lần thử thành

công xảy ra với xác suất p và lần thất bại xảy ra với xác suất $q = 1 - p$. Thì với $r > np$,

$$\begin{aligned}\Pr\{X - np \geq r\} &= \sum_{k=\lceil np+r \rceil}^{n} b(k; n, p) \\ &\leq \left(\frac{npe}{r}\right)^r .\end{aligned}$$

Chứng minh

Với một phép phân phối nhị thức, phương trình (6.39) hàm ý $\mu = \mathrm{E}[X] = np$.

Bài tập

6.5-1 *

Cái nào ắt nhỏ hơn: có được không mặt ngửa khi bạn tung một đồng tiền cân n lần, hoặc có được ít hơn n mặt ngửa khi bạn tung đồng tiền $4n$ lần?

6.5-2 *

Chứng tỏ

$$\sum_{i=0}^{k-1} \binom{n}{i} a^i < (a+1)^n \frac{k}{na - k(a+1)} b(k; n, a/(a+1))$$

với tất cả $a > 0$ và tất cả k sao cho $0 < k < n$.

6.5-3 *

Chứng minh rằng nếu $0 < k < np$, ở đó $0 < p < 1$ và $q = 1 - p$, thì

$$\sum_{i=0}^{k-1} p^i q^{n-i} < \frac{kq}{np - k} \left(\frac{np}{k}\right)^k \left(\frac{nq}{np - k}\right)^{n-k} .$$

6.5-4 *

Chứng tỏ các điều kiện của Định lý 6.6 hàm ý rằng

$$\Pr\{\mu - X \geq r\} \leq \left(\frac{(n-\mu)e}{r}\right)^r .$$

Cũng vậy, chứng tỏ các điều kiện của Hệ luận 6.7 hàm ý rằng

$$\Pr\{np - X \geq r\} \leq \left(\frac{npe}{r}\right)^r .$$

6.5-5 *

Xem xét một dãy n lần thử Bernoulli, ở đó trong lần thử thứ i, với $i = 1, 2, \ldots, n$, lần thành công xảy ra với xác suất p_i và lần thất bại xảy ra với

xác suất $q_i = 1 - p_i$. Cho X là biến ngẫu nhiên mô tả tổng số lần thành công, và cho $\mu = E[X]$. Chứng tỏ với $r \geq 0$,

$\Pr\{X - \mu \geq r\} \leq e^{-r^2/2n}$,

(*Mách nước:* Chứng minh rằng $p_i e^{-\alpha q_i} + q_i e^{-\alpha p_i} \leq e^{-\alpha^2/2}$. Sau đó theo đề cương của phép chứng minh Định lý 6.6, dùng bất đẳng thức này thay vì bất đẳng thức (6.44).)

6.5-6 *

Chứng tỏ việc chọn $\alpha = \ln(r/\mu)$ giảm thiểu bên phía phải của bất đẳng thức (6.45).

6.6 Phân tích xác suất

Đoạn này sử dụng ba ví dụ để minh họa tiến trình phân tích xác suất. Ví dụ đầu tiên xác định xác suất rằng, trong một phòng có k người, một cặp nào đó chia sẻ cùng ngày sinh nhật. Ví dụ thứ hai xem xét việc tung ngẫu nhiên các quả bóng vào các giỏ. Ví dụ thứ ba nghiên cứu "các vệt" của các mặt ngửa liên tục trong khi tung đồng tiền.

6.6-1 Nghịch lý ngày sinh nhật

Một ví dụ tốt để minh họa tiến trình biện luận xác suất đó là ***nghịch lý ngày sinh nhật*** cổ điển. Phải có bao nhiêu người trong một phòng trước khi có một cơ may tốt ở đó hai trong số họ sinh cùng ngày trong năm? Câu trả lời quả ít đến kinh ngạc. Nghịch lý đó là nó thực tế ít hơn nhiều so với số ngày trong năm, như sẽ thấy.

Để trả lời câu hỏi, ta lập chỉ số các người trong phòng với các số nguyên 1, 2,..., k, ở đó k là số lượng người trong phòng. Ta bỏ qua vấn đề năm nhuận và mặc nhận rằng tất cả các năm đều có $n = 365$ ngày. Với $i = 1, 2, \ldots, k$, cho b_i là ngày trong năm mà ngày sinh nhật của i rơi trúng, ở đó $1 \leq b_i \leq n$. Ta cũng mặc nhận rằng các ngày sinh nhật được phân phối đều qua n ngày trong năm, sao cho $\Pr\{b_i = r\} = 1/n$ với $i = 1, 2, \ldots, k$ và $r = 1, 2, \ldots, n$.

Xác suất mà hai người i và j có các ngày sinh nhật so khớp sẽ tùy thuộc vào việc lựa chọn ngẫu nhiên các ngày sinh nhật có độc lập hay không. Nếu các ngày sinh nhật độc lập, thì xác suất mà ngày sinh nhật của i lẫn ngày sinh nhật của j rơi trúng ngày r là

$$\begin{aligned}\Pr\{b_i = r \text{ và } b_j = r\} &= \Pr\{b_i = r\}\Pr\{b_j = r\} \\ &= 1/n^2 .\end{aligned}$$

Như vậy, xác suất mà cả hai rơi vào cùng ngày là

$$\begin{aligned}\Pr\{b_i = b_j\} &= \sum_{r=1}^{n} \Pr\{b_i = r \text{ và } b_j = r\} \\ &= \sum_{r=1}^{n} (1/n^2) \\ &= 1/n .\end{aligned}$$

Trực giác hơn, một khi đã chọn b_i, xác suất mà b_j được chọn cũng giống như $1/n$. Như vậy, xác suất mà i và j có cùng ngày sinh nhật cũng giống như xác suất mà ngày sinh nhật của một hai rơi vào một ngày đã cho. Tuy nhiên, lưu ý sự trùng hợp này tùy thuộc vào giả thiết các ngày sinh nhật là độc lập.

Ta có thể phân tích xác suất của ít nhất 2 trong số k người có các ngày sinh nhật so khớp bằng cách xem xét sự kiện bổ sung. Xác suất mà ít nhất hai trong số các ngày sinh nhật so khớp đó là 1 trừ xác suất mà tất cả các ngày sinh nhật khác nhau. Sự kiện mà k người có các ngày sinh nhật riêng biệt là

$$B_k = \bigcap_{i=1}^{k-1} A$$

ở đó A_i là sự kiện mà ngày sinh nhật của cá nhân $(i + 1)$ khác với cá nhân j với tất cả $j \le i$, nghĩa là,

$$A_i = \{b_{i+1} \ne b_j : j = 1, 2, ..., i\} .$$

Do có thể viết $B_k = A_{k-1} \cap B_{k-1}$, nên ta có từ phương trình (6.20) phép truy toán

$$\Pr\{B_k\} = \Pr\{B_{k-1}\} \Pr\{A_{k-1} \mid B_{k-1}\} \qquad (6.46)$$

ở đó ta chọn $\Pr\{B_1\} = 1$ làm một điều kiện ban đầu. Nói cách khác, xác suất mà $b_1, b_2, ..., b_k$ là các ngày sinh nhật riêng biệt chính là xác suất mà $b_1, b_2, ..., b_{k-1}$, là các ngày sinh nhật riêng biệt nhân với xác suất mà $b_k \ne b_i$ với $i = 1, 2, ..., k - 1$, căn cứ vào $b_1, b_2, ..., b_{k-1}$ là riêng biệt.

Nếu $b_1, b_2, ..., b_{k-1}$ là riêng biệt, xác suất có điều kiện mà $b_k \ne b_i$ với $i = 1, 2, ..., k - 1$ là $(n - k + 1)/n$, bởi từ n ngày, có $n - (k - 1)$ không được chọn. Nhờ lặp lại phép truy toán (6.46), ta được

$$\begin{aligned}\Pr\{B_k\} &= \Pr\{B_1\}\Pr\{A_1 \mid B_1\}\Pr\{A_2 \mid B_2\} \dots \Pr\{A_{k-1} \mid B_{k-1}\} \\ &= 1 \cdot \left(\frac{n-1}{n}\right)\left(\frac{n-2}{n}\right) \cdots \left(\frac{n-k+1}{n}\right) \\ &= 1 \cdot \left(1 - \frac{1}{n}\right)\left(1 - \frac{2}{n}\right) \cdots \left(1 - \frac{k-1}{n}\right) .\end{aligned}$$

Bất đẳng thức (2.7), $1 + x \le e^x$, cho ta

$$\begin{aligned}\Pr\{B_k\} &\le e^{-1/n}e^{-2/n}\ldots e^{-(k-1)/n}\\ &= e^{-\sum_{i=1}^{k-1} i/n}\\ &= e^{-k(k-1)/2n}\\ &\le 1/2\end{aligned}$$

khi $-k(k-1)/2n \le \ln(1/2)$. Xác suất mà tất cả k ngày sinh nhật là riêng biệt sẽ tối đa là 1/2 khi $k(k-1) \ge 2n \ln 2$ hoặc, giải phương trình bậc hai, khi $k \ge (1 + 1 + \sqrt{8\ln 2)n})/2$. Với $n = 365$, ta phải có $k \ge 23$. Như vậy, nếu có ít nhất 23 người trong một phòng, xác suất ít nhất là 1/2 rằng ít nhất hai người có cùng ngày sinh nhật. Trên sao Hỏa, một năm dài 669 ngày sao Hỏa; do đó cần có 31 người sao Hỏa để có cùng hiệu ứng.

Một phương pháp phân tích khác

Ta có thể dùng tính chất tuyến tính của giá trị kỳ vọng (phương trình (6.26)) để cung cấp một phương pháp phân tích tuy đơn giản hơn song xấp xỉ về nghịch lý ngày sinh nhật. Với mỗi cặp (i, j) trong số k người trong phòng, ta định nghĩa biến ngẫu nhiên X_{ij}, với $1 \le i < j \le k$, theo

$$X_{ij} = \begin{cases} 1 & \text{nếu cá nhân } i \text{ và cá nhân } j \text{ có cùng ngày sinh nhật },\\ 0 & \text{bằng không.}\end{cases}$$

Xác suất mà hai người có các ngày sinh nhật so khớp là $1/n$, và như vậy theo định nghĩa về giá trị kỳ vọng (6.23),

$$\begin{aligned}\mathrm{E}[X_{ij}] &= 1 \cdot (1/n) + 0 \cdot (1-1/n)\\ &= 1/n\,.\end{aligned}$$

Số lượng dự trù các cặp cá nhân có cùng ngày sinh nhật đơn giản là, theo phương trình (6.24), tổng các giá trị kỳ vọng riêng lẻ của các cặp, là

$$\begin{aligned}\sum_{i=2}^{k}\sum_{j=1}^{i-1}\mathrm{E}[X_{ij}] &= \binom{k}{2}\frac{1}{n}\\ &= \frac{k(k-1)}{2n}\,.\end{aligned}$$

Do đó, khi $k(k-1) \ge 2n$, số lượng dự trù các cặp ngày sinh nhật ít nhất là 1. Như vậy, nếu ta có ít nhất $\sqrt{2n}$ cá nhân trong một phòng, ta có thể dự trù ít nhất hai người có cùng ngày sinh nhật. Với $n = 365$, nếu $k = 28$, số lượng dự trù các cặp có cùng ngày sinh nhật là $(28 \,.\, 27)/(2 \,.\, 365) \approx 1.0356$. Như vậy, Với ít nhất 28 người, ta dự trù tìm ra ít nhất một cặp các ngày sinh nhật so khớp. Trên sao Hỏa, một năm dài 669 ngày sao Hỏa; ta cần ít nhất 38 người Hỏa để có cùng hiệu ứng.

Phân tích đầu tiên đã xác định số lượng người cần thiết cho xác suất vượt quá 1/2 ở đó tồn tại một cặp ngày sinh nhật so khớp, và phân tích thứ hai đã xác định con số sao cho số lượng dự trù các ngày sinh nhật so khớp bằng 1. Mặc dù các số người khác nhau trong hai tình huống, chúng là như nhau theo tiệm cận: $\Theta(\sqrt{n})$.

6.6.2 Các quả bóng và các giỏ

Xem xét tiến trình thảy ngẫu nhiên các quả bóng giống nhau vào b giỏ, được đánh số 1, 2,..., b. Các lần thảy là độc lập, và với mỗi lần thảy, quả bóng có khả năng kết thúc bằng nhau trong một giỏ bất kỳ. Xác suất mà một quả bóng đã thảy đáp vào một giỏ bất kỳ đã cho là $1/b$. Như vậy, tiến trình thảy bóng là một dãy các lần thử Bernoulli với một xác suất $1/b$ thành công, ở đó thành công có nghĩa là quả bóng rơi vào giỏ đã cho. Có thể đặt ra nhiều câu hỏi thú vị về tiến trình thảy bóng.

Bao nhiêu quả bóng rơi vào một giỏ đã cho? Số lượng quả bóng rơi vào một giỏ đã cho sẽ theo phép phân phối nhị thức $b(k;n, 1/b)$. Nếu n quả bóng được thảy, số lượng quả bóng dự trù rơi vào giỏ đã cho là n/b.

Trung bình, phải thảy bao nhiêu quả bóng, cho đến khi một giỏ đã cho chứa một quả bóng? Số lượng lần thảy cho đến khi giỏ đã cho nhận được một quả bóng sẽ theo phép phân phối cấp số nhân với xác suất $1/b$, và như vậy số lần thảy dự trù cho đến khi thành công là $1/(1/b) = b$.

Phải thảy bao nhiêu quả bóng cho đến khi mọi giỏ chứa ít nhất một quả bóng?

Ta gọi một lần thảy ở đó một quả bóng rơi vào một giỏ trống là một "lần ném trúng." Ta muốn biết số trung bình n lần thảy cần thiết để có b lần ném trúng.

Các lần ném trúng có thể được dùng để phân hoạch n lần thảy thành các giai đoạn. Giai đoạn thứ i bao gồm các lần thảy sau lần ném trúng $(i - 1)$ cho đến lần ném trúng thứ i. Giai đoạn đầu tiên gồm cả lần thảy đầu tiên, bởi ta được bảo đảm có một lần ném trúng khi tất cả các giỏ là trống. Với mỗi lần thảy trong giai đoạn thứ i, có $i - 1$ giỏ chứa các quả bóng và $b - i + 1$ các giỏ trống. Như vậy, với tất cả các lần thảy trong giai đoạn thứ i, xác suất có được một lần ném trúng là $(b - i + 1)/b$.

Cho n_i thể hiện số lượng các lần thảy trong giai đoạn thứ i. Như vậy, số lượng các lần thảy cần thiết để có b lần ném trúng là $n = \sum_{i=1}^{b} n_i$. Mỗi biến ngẫu nhiên n_i có một phép phân phối cấp số nhân với xác suất thành công $(b - i + 1)/b$, và do đó

$$E[n_i] = \frac{b}{b - i + 1} .$$

Theo tính chất tuyến tính của giá trị kỳ vọng,

$$\begin{aligned} E[n] &= E\left[\sum_{i=1}^{b} n_i\right] \\ &= \sum_{i=1}^{b} E[n_i] \\ &= \sum_{i=1}^{b} \frac{b}{b - i + 1} \\ &= b\sum_{i=1}^{b} \frac{1}{i} \\ &\leq b(\ln b + O(1)) . \end{aligned}$$

Dòng cuối là do cận (3.5) trên chuỗi điều hòa. Do đó ta phải có xấp xỉ $b \ln b$ lần thảy trước khi có thể mong đợi mọi giỏ có một quả bóng.

6.6.3 Các vệt

Giả sử bạn tung một đồng tiền cân n lần. Đâu là vệt [streak] dài nhất của các mặt ngửa liên tiếp mà bạn dự kiến sẽ gặp? Câu trả lời là $\Theta(\lg n)$, như tiến trình phân tích dưới đây cho thấy.

Trước tiên ta chứng minh chiều dài dự trù vệt dài nhất của các mặt ngửa là $O(\lg n)$. Cho A_{ik} là sự kiện mà một vệt của các mặt ngửa có chiều dài ít nhất k bắt đầu bằng lần tung đồng tiền thứ i hoặc, chính xác hơn, là sự kiện mà k lần tung đồng tiền liên tục $i, i + 1, \ldots, i + k - 1$ chỉ cho ra các mặt ngửa, ở đó $1 \leq k \leq n$ và $1 \leq i \leq n - k + 1$. Với bất kỳ sự kiện đã cho A_{ik}, xác suất mà tất cả k lần tung là các mặt ngửa có một phép phân phối cấp số nhân với $p = q = 1/2$:

$$\Pr\{A_{ik}\} = 1/2^k . \qquad (6.47)$$

For $k = 2\lceil \lg n \rceil$,

$$\begin{aligned} \Pr\{A_{i,2\lceil \lg n \rceil}\} &= 1/2^{2\lceil \lg n \rceil} \\ &\leq 1/2^{2\lg n} \\ &= 1/n^2 , \end{aligned}$$

và như vậy xác suất mà một vệt của các mặt ngửa có chiều dài ít nhất $2\lceil \lg n \rceil$ bắt đầu tại vị trí i là khá nhỏ, nhất là xét thấy có tối đa n vị trí (thực tế $n - 2\lceil \lg n \rceil + 1$) ở đó vệt có thể bắt đầu. Do đó, xác suất mà một vệt của các mặt ngửa có chiều dài ít nhất $\lceil 2\lg n \rceil$ bắt đầu tại bất kỳ đâu là

$$\Pr\left\{\bigcup_{i=1}^{n-2\lceil \lg n\rceil+1} A_{i,2\lceil \lg n\rceil}\right\} \le \sum_{i=1}^{n} 1/n^2$$
$$= 1/n ,$$

bởi theo bất đẳng thức Bool (6.22), xác suất của một hợp các sự kiện tối đa là tổng các xác suất của các sự kiện riêng lẻ. (Lưu ý, bất đẳng thức Bool vẫn có hiệu lực kể cả với các sự kiện không độc lập như ở đây.)

Do đó xác suất tối đa là $1/n$ mà một vệt bất kỳ có chiều dài ít nhất $2\lceil \lg n\rceil$; do đó, xác suất ít nhất là $1 - 1/n$ mà vệt dài nhất có chiều dài nhỏ hơn $2\lceil \lg n\rceil$. Bởi mọi vệt đều có chiều dài tối đa n, nên chiều dài dự trù của vệt dài nhất được định cận bên trên theo

$$(2\lceil \lg n\rceil)(1 - 1/n) + n(1/n) = O(\lg n) .$$

Các cơ may mà một vệt của các mặt ngửa vượt quá $r\lceil \lg n\rceil$ lần tung sẽ giảm nhanh với r. Với $r \ge 1$, xác suất mà một vệt của $r\lceil \lg n\rceil$ mặt ngửa bắt đầu tại vị trí i là

$$\Pr\{A_{i,r\lceil \lg n\rceil}\} = 1/2^{r\lceil \lg n\rceil}$$
$$\le 1/n^r .$$

Như vậy, xác suất tối đa là $n/n^r = 1/n^{r-1}$ mà vệt dài nhất ít ra là $r\lceil \lg n\rceil$, hoặc tương đương, xác suất ít nhất là $1 - 1/n^{r-1}$ mà vệt dài nhất có chiều dài nhỏ hơn $r\lceil \lg n\rceil$.

Để lấy ví dụ, với $n = 1000$ lần tung đồng tiền, xác suất có một vệt ít nhất $2\lceil \lg n\rceil = 20$ mặt ngửa tối đa là $1/n = 1/1000$. Các cơ may có một vệt dài hơn $3\lceil \lg n\rceil = 30$ mặt ngửa tối đa là $1/n^2 = 1/1,000,000$.

Giờ đây ta chứng minh một cận dưới bổ sung: chiều dài dự trù vệt dài nhất của các mặt ngửa trong n lần tung đồng tiền là $\Omega(\lg n)$. Để chứng minh cận này, ta tìm các vệt có chiều dài $\lceil \lg n\rceil/2$. Từ phương trình (6.47), ta có

$$\Pr\{A_{i,\lfloor \lg n\rfloor/2}\} = 1/2^{\lfloor \lg n\rfloor/2}$$
$$\ge 1/\sqrt{n} .$$

Do đó xác suất mà một vệt của các mặt ngửa có chiều dài ít nhất $\lfloor \lg n\rfloor/2$ không bắt đầu tại vị trí i sẽ tối đa là $1 - 1/\sqrt{n}$. Ta có thể phân hoạch n lần tung đồng tiền thành ít nhất $\lfloor 2n/\lfloor \lg n\rfloor\rfloor$ nhóm của $\lfloor \lg n\rfloor/2$ lần tung đồng tiền liên tục. Bởi các nhóm này được hình thành từ các lần tung đồng tiền độc lập, loại trừ nhau, nên xác suất mà mỗi một nhóm này *thất bại* để trở thành một vệt có chiều dài $\lfloor \lg n\rfloor/2$ sẽ là

$$(1 - 1/\sqrt{n})^{\lfloor 2n/\lfloor \lg n\rfloor\rfloor} \le (1 - 1/\sqrt{n})^{2n/\lg n - 1}$$

$$\leq e^{-(2n/\lg n - 1)/\sqrt{n}}$$

$$\leq e^{-\lg n}$$

$$\leq 1/n .$$

Với biện luận này, ta đã dùng bất đẳng thức (2.7), $1 + x \leq e^x$, và sự việc (mà bạn phải xác minh) cho rằng $(2n/\lg n - 1)/\sqrt{n} \geq \lg n$ với $n \geq 2$. (Với $n = 1$, xác suất mà mọi nhóm thất bại trở thành một vệt tối đa không đáng kể là $1/n = 1$.)

Như vậy, xác suất mà vệt dài nhất vượt quá $\lfloor \lg n \rfloor /2$ ít nhất là $1 - 1/n$. Bởi vệt dài nhất có chiều dài ít nhất 0, nên chiều dài dự trù của vệt dài nhất sẽ ít nhất là

$$(\lfloor \lg n \rfloor /2)(1 - 1/n) + 0 \cdot (1/n) = \Omega(\lg n) .$$

Bài tập

6.6-1

Giả sử các quả bóng được thảy vào *b* giỏ. Mỗi lần thảy là độc lập, và mỗi quả bóng có khả năng kết thúc bằng nhau trong một giỏ bất kỳ. Đâu là số lần thảy bóng dự trù trước khi ít nhất một trong các giỏ chứa hai quả bóng?

6.6-2 *

Với kiểu phân tích nghịch lý ngày sinh nhật, các ngày sinh nhật độc lập tương hỗ là quan trọng, hay chỉ cần sự độc lập theo từng cặp là đủ? Xác minh câu trả lời.

6.6-3 *

Sẽ phải mời bao nhiêu người dự tiệc để ắt có *ba* người có cùng ngày sinh nhật?

6.6-4 *

Đâu là xác suất mà một chuỗi-*k* trên một tập hợp có kích cỡ *n* thực tế là một phép hoán vị-*k*? Câu hỏi này liên quan như thế nào với nghịch lý ngày sinh nhật?

6.6-5 *

Giả sử *n* quả bóng được thảy vào *n* giỏ, ở đó mỗi lần thảy là độc lập và quả bóng có khả năng kết thúc như nhau trong một giỏ bất kỳ. Đâu là số lượng các giỏ trống dự trù? Đâu là số lượng giỏ dự trù có chính xác một quả bóng?

6.6-6 *

Tinh chỉnh cận dưới dựa trên chiều dài vệt [streak] bằng cách chứng

tỏ trong n lần tung một đồng tiền cân, xác suất nhỏ hơn $1/n$ mà không xảy ra vệt nào dài hơn $\lg n - 2 \lg \lg n$ mặt ngửa liên tục.

Các Bài Toán

6-1 Các quả bóng và các giỏ

Trong bài toán này, ta nghiên cứu hiệu ứng của các giả thiết khác nhau về số cách đặt n quả bóng vào b giỏ riêng biệt.

a. Giả sử n quả bóng là riêng biệt và thứ tự của chúng trong một giỏ là không thành vấn đề. Chứng tỏ số cách đặt các quả bóng trong các giỏ là b^n.

b. Giả sử các quả bóng là riêng biệt và các quả bóng trong mỗi giỏ được sắp xếp thứ tự. Chứng minh rằng số cách đặt các quả bóng trong các giỏ là $b + n - 1)!/(b - 1)!$. (*Mách nước.* Xem xét số cách dàn xếp n quả bóng riêng biệt và $b - 1$ đợt không phân biệt được trong một hàng.)

c. Giả sử các quả bóng giống nhau, và do đó thứ tự của chúng trong một giỏ là không thành vấn đề. Chứng tỏ số cách đặt các quả bóng trong các giỏ là $\binom{b+n-1}{n}$. (*Mách nước:* Từ các dàn xếp trong phần (b), được lặp lại bao nhiêu nếu các quả bóng được làm giống nhau?)

d. Giả sử các quả bóng giống nhau và không giỏ nào có thể chứa nhiều hơn một quả bóng. Chứng tỏ số cách đặt các quả bóng là $\binom{b}{n}$.

e. Giả sử các quả bóng giống nhau và không giỏ nào có thể để trống. Chứng tỏ số cách đặt các quả bóng là $\binom{n-1}{b-1}$.

6.2 Phân tích chương trình max

Chương trình dưới đây xác định giá trị cực đại trong một mảng không sắp xếp thứ tự $A[1..n]$.

```
1 max ← -∞
2 for i ← 1 to n
3       do ▷ Compare A[i] to max.
4           If A[i] > max
5               Then max ← A[i]
```

Ta muốn xác định số lần trung bình thi hành phép gán trong dòng 5. Giả sử tất cả các số trong A được rút ngẫu nhiên từ khoảng $[0, 1]$.

a. Nếu một số x được chọn ngẫu nhiên từ một tập hợp n số riêng biệt, đâu là xác suất mà x là số lớn nhất trong tập hợp?

b. Khi dòng 5 của chương trình được thi hành, đâu là quan hệ giữa $A[i]$ và $A[j]$ với $1 \leq j \leq i$?

c. Với mỗi i trong miền giá trị $1 \leq i \leq n$, đâu là xác suất mà dòng 5 được thi hành?

d. Cho $s_1, s_2, ..., s_n$ là n biến ngẫu nhiên, ở đó s_i biểu thị cho số lần (0 hoặc 1) mà dòng 5 được thi hành trong lần lặp thứ i của vòng lặp **for**. E $[s_i]$ là gì?

e. Cho $s = s_1 + s_2 + \cdots + s_n$ là tổng số lần mà dòng 5 được thi hành trong một lần chạy nào đó của chương trình. Chứng tỏ E $[s] = \Theta(\lg n)$.

6-3 Bài toán thuê

Giáo sư Dixon cần thuê một viên phụ tá nghiên cứu mới. Giáo sư đã dàn xếp các cuộc phỏng vấn với n người xin việc và chỉ muốn quyết định dựa trên trình độ chuyên môn của họ. Đáng tiếc, các quy định của đại học yêu cầu sau mỗi cuộc phỏng vấn Giáo sư phải lập tức loại hoặc giao chức vụ cho người xin việc.

Giáo sư Dixon quyết định chấp nhận chiến lược lựa một số nguyên dương $k < n$, phỏng vấn rồi loại k người xin việc đầu tiên, và sau đó thuê người xin việc đầu tiên có năng lực tốt hơn tất cả các người xin việc trước đó. Nếu người xin việc có năng lực tốt nhất nằm trong k người được phỏng vấn đầu tiên, thì giáo sư sẽ thuê người xin việc thứ n. Chứng tỏ Giáo sư Dixon tối đa hóa các cơ may của mình để thuê người xin việc có năng lực nhất bằng cách chọn k xấp xỉ bằng với n/e và cơ may thuê được người xin việc có năng lực nhất sẽ xấp xỉ là $1/e$.

6-4 Đếm xác suất

Với một bộ đếm t-bit, ta thường chỉ có thể đếm tới $2^t - 1$. Với ***phép đếm xác suất*** của R. Morris, ta có thể đếm tới một giá trị lớn hơn nhiều nhưng đổi lại có phần thiếu chính xác.

Cho một giá trị bộ đếm i biểu thị cho một số đếm của n_i với $i = 0, 1, ..., 2^t - 1$, ở đó n_i hình thành một dãy tăng các giá trị không âm. Ta mặc nhận rằng giá trị ban đầu của bộ đếm là 0, biểu thị một số đếm của $n_0 = 0$. Phép toán INCREMENT làm việc trên một bộ đếm chứa giá trị i theo kiểu xác suất. Nếu $i = 2^t - 1$, thì một lỗi tràn được báo cáo. Bằng không, bộ đếm được tăng theo 1 với xác suất $1/(n_{i+1} - n_i)$, và nó giữ nguyên không đổi với xác suất $1 - 1/(n_{i+1} - n_i)$.

Nếu ta lựa $n_i = i$ với tất cả $i \geq 0$, thì bộ đếm là một bộ đếm bình thường. Các tình huống thú vị hơn sẽ nảy sinh nếu ta lựa, giả sử, $n_i = 2^{i-1}$ với $i > 0$ hoặc $n_i = F_i$ (số Fibonacci thứ i—xem Đoạn 2.2).

Với bài toán này, giả sử $n_{2^{l}-1}$ đủ lớn để xác suất của một lỗi tràn là không đáng kể.

a. Chứng tỏ giá trị dự trù được biểu thị bởi bộ đếm sau khi thực hiện n phép toán INCREMENT sẽ chính xác là n.

b. Việc phân tích phương sai của số đếm được biểu thị bởi bộ đếm sẽ tùy thuộc vào dãy n_i. Ta hãy xét một trường hợp đơn giản: $n_i = 100i$ với tất cả $i \geq 0$. Đánh giá phương sai trong giá trị mà thanh ghi biểu thị sau khi thực hiện n phép toán INCREMENT.

Ghi chú Chương

Các phương pháp chung đầu tiên để giải các bài toán xác suất đã được đề cập trong tài liệu trao đổi nổi tiếng giữa B. Pascal và P. de Fermat, bắt đầu vào năm 1654, và trong một cuốn sách của C. Huygens vào năm 1657. Lý thuyết xác suất chính xác đã bắt đầu với công trình của J. Bernoulli năm 1713 và A. De Moivre năm 1730. Lý thuyết đã được P. S. de Laplace, S.-D. Poisson, và C. F. Gauss phát triển thêm.

Các tổng của các biến ngẫu nhiên đã được P. L. Chebyshev và A. A. Markov nghiên cứu đầu tiên. Lý thuyết xác suất đã được A. N. Kolmogorov tiên đề hóa năm 1933. Các cận trên đệ quy đuôi của các phép phân phối đã được cung cấp Chernoff [40] và Hoeffding [99] cung cấp. P. Erdos đã thực hiện công trình hạt giống về các cấu trúc tổ hợp ngẫu nhiên.

Knuth [121] và Liu [140] là những nguồn tham khảo tốt về toán học tổ hợp sơ cấp và phép đếm. Các tài liệu giáo khoa chuẩn như Billingsley [28], Chung [41], Drake [57], Feller [66], và Rozanov [171] đã giới thiệu toàn diện về xác suất. Bollobas [30], Hofri [100], và Spencer [179] đề cập chi tiết các kỹ thuật xác suất cao cấp.

II Sắp Xếp và Thống Kê Thứ Tự

Nhập đề

Phần này trình bày vài thuật toán giải ***bài toán sắp xếp*** dưới đây:

Đầu vào: Một dãy *n* số <a_1, a_2,..., a_n>.

Kết xuất: Một phép hoán vị (sắp xếp lại thứ tự) <a'_1, a'_2,..., a'_n> dãy đầu vào sao cho $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Dãy đầu vào thường là một mảng *n*-thành phần, mặc dù nó có thể được biểu thị theo vài cách khác, như một danh sách nối kết.

Cấu trúc dữ liệu

Trong thực tế, các con số để sắp xếp hiếm khi là những giá trị cô lập. Mỗi giá trị thường là một phần của một tập hợp dữ liệu có tên ***khoản tin*** [record]. Mỗi khoản tin chứa một ***khóa*** [key], là giá trị sẽ được sắp xếp, và phần còn lại của khoản tin bao gồm ***dữ liệu vệ tinh*** [satellite data], thường được mang đi với khóa. Trong thực tế, khi một thuật toán sắp xếp hoán vị các khóa, nó cũng phải hoán vị dữ liệu vệ tinh. Nếu mỗi khoản tin gộp một lượng lớn dữ liệu vệ tinh, ta thường hoán vị một mảng các biến trỏ đến các khoản tin thay vì chính các khoản tin đó để giảm thiểu việc dời chuyển dữ liệu.

Theo một nghĩa nào đó, chính các chi tiết thực thi này sẽ phân biệt một thuật toán với một chương trình hoàn chỉnh. Dẫu sắp xếp các con số riêng lẻ hay các khoản tin lớn chứa các con số không liên quan đến phương pháp mà một thủ tục sắp xếp dùng để xác định thứ tự sắp xếp. Như vậy, khi tập trung vào bài toán sắp xếp, ta thường mặc nhận đầu vào chỉ gộp các con số. Việc phiên dịch của một thuật toán để sắp xếp các con số thành một chương trình để sắp xếp các khoản tin thường dễ hiểu về khái niệm, mặc dù trong một tình huống thiết kế kỹ thuật nhất định, có thể có các điểm tinh tế khác khiến việc lập trình thực tế trở thành một thách thức.

Các thuật toán sắp xếp

Chương 1 đã giới thiệu hai thuật toán sắp xếp n số thực. Sắp xếp chèn mất $\Theta(n^2)$ thời gian trong trường hợp xấu nhất. Tuy nhiên, do các vòng lặp trong của nó chặt chẽ, nên nó là một thuật toán sắp xếp tại chỗ nhanh đối với các kích cỡ đầu vào nhỏ. (Chắc bạn còn nhớ, một thuật toán sắp xếp sắp xếp ***tại chỗ*** [in place] nếu chỉ một số lượng thành phần bất biến của mảng đầu vào được sắp xếp bên ngoài mảng.) Sắp xếp trộn có thời gian thực hiện tiệm cận tốt hơn, $\Theta(n \lg n)$, nhưng thủ tục MERGE mà nó sử dụng lại không hoạt động tại chỗ.

Phần này sẽ giới thiệu thêm hai thuật toán sắp xếp các số thực tùy ý. Sắp xếp đống [heapsort], được trình bày trong Chương 7, sẽ sắp xếp n con số tại chỗ trong thời gian $O(n \lg n)$. Nó sử dụng một cấu trúc dữ liệu quan trọng, có tên ***đống*** [heap], để thực thi một hàng đợi ưu tiên.

Sắp xếp nhanh [quicksort], trong Chương 8, cũng sắp xếp n con số tại chỗ, nhưng thời gian thực hiện trường hợp xấu nhất của nó là $\Theta(n^2)$. Tuy vậy, thời gian thực hiện trường hợp trung bình của nó là $\Theta(n \lg n)$, và trong thực tế nó thường thực hiện tốt hơn kỹ thuật sắp xếp đống. Giống như sắp xếp chèn, sắp xếp nhanh cũng có mã chặt chẽ, do đó thừa số bất biến ẩn trong thời gian thực hiện của nó là nhỏ. Nó là một thuật toán phổ dụng để sắp xếp các mảng đầu vào lớn.

Sắp xếp chèn, sắp xếp trộn, sắp xếp đống, và sắp xếp nhanh, tất cả đều là kiểu sắp xếp so sánh: chúng xác định thứ tự sắp xếp của một mảng đầu vào bằng cách so sánh các thành phần. Đầu Chương 9 sẽ giới thiệu mô hình cây-quyết định để nghiên cứu các hạn chế về khả năng thực hiện của các kiểu sắp xếp so sánh. Dùng mô hình này, ta chứng minh một cận dưới của $\Omega(n \lg n)$ trên thời gian thực hiện ca xấu nhất của bất kỳ kiểu sắp xếp so sánh nào trên n đầu vào, như vậy chứng tỏ rằng sắp xếp đống và sắp xếp trộn là các kiểu sắp xếp so sánh tối ưu theo tiệm cận.

Chương 9 tiếp tục chứng tỏ ta có thể đánh bại cận dưới này của $\Omega(n \lg n)$ nếu như có thể thu thập thông tin về thứ tự sắp xếp của đầu vào bằng các biện pháp khác ngoài các thành phần so sánh. Ví dụ, thuật toán sắp xếp đếm [counting sort] mặc nhận các con số đầu vào nằm trong tập hợp $\{1, 2, \ldots, k\}$. Nhờ dùng tính năng lập chỉ mục mảng làm công cụ để xác định thứ tự tương đối, sắp xếp đếm có thể sắp xếp n con số trong $O(k + n)$ thời gian. Như vậy, khi $k = O(n)$, sắp xếp đếm chạy trong thời gian tuyến tính theo kích cỡ mảng đầu vào. Một thuật toán tương đối, sắp xếp cơ số [radix sort], có thể được dùng để khai triển miền giá trị của sắp xếp đếm. Nếu có n số nguyên để sắp xếp, mỗi số nguyên có d chữ số, và mỗi chữ số nằm trong tập hợp $\{1, 2, \ldots, k\}$, sắp

xếp cơ số có thể sắp xếp các con số trong $O(d(n + k))$ thời gian. Khi d là một hằng và k là $O(n)$, sắp xếp cơ số sẽ chạy trong thời gian tuyến tính. Một thuật toán thứ ba, sắp xếp thùng [bucket sort], yêu cầu am hiểu phép phân phối xác suất các con số trong mảng đầu vào. Nó có thể sắp xếp n số thực được phân phối đều trong khoảng nửa-mở $[0,1)$ trong $O(n)$ thời gian trường hợp trung bình.

Thống kê thứ tự

Thống kê thứ tự thứ i của một tập hợp n số là con số nhỏ nhất thứ i trong tập hợp. Tất nhiên, ta có thể lựa thống kê thứ tự thứ i bằng cách sắp xếp đầu vào và lập chỉ mục thành phần thứ i của kết xuất. Nếu không có các giả thiết về phép phân phối đầu vào, phương pháp này chạy trong $\Omega(n \lg n)$ thời gian, như cận dưới được chứng minh trong Chương 9 cho thấy.

Chương 10 cho thấy ta có thể tìm ra thành phần nhỏ nhất thứ i trong $O(n)$ thời gian, kể cả khi các thành phần là các số thực tùy ý. Chương cũng trình bày một thuật toán với mã giả chặt chạy trong $O(n^2)$ thời gian trong trường hợp xấu nhất, nhưng thời gian tuyến tính tính trung bình. Ngoài ra còn có một thuật toán phức hợp hơn chạy trong $O(n)$ thời gian trường hợp xấu nhất.

Kiến thức căn bản

Mặc dù hầu hết phần này không dựa vào khái niệm toán học khó, song có vài đoạn yêu cầu biện luận phức hợp về toán học. Nhất là, các tiến trình phân tích trường hợp trung bình của thuật toán sắp xếp nhanh, sắp xếp thùng [bucket sort], và thống kê thứ tự sử dụng xác suất, đã được ôn lại trong Chương 6. Tiến trình phân tích của thuật toán thời gian tuyến tính trường hợp xấu nhất với thống kê thứ tự thường liên quan đến khái niệm toán học hơi phức tạp hơn so với các tiến trình phân tích trường hợp xấu nhất khác trong phần này.

7 Sắp Xếp Đống

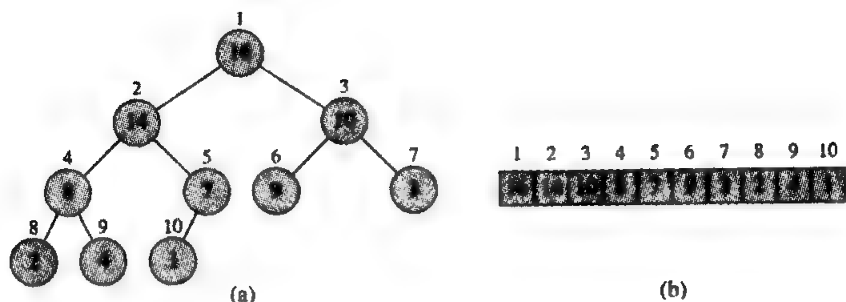
Chương này giới thiệu một thuật toán sắp xếp khác. Giống như sắp xếp trộn, nhưng khác với sắp xếp chèn, thời gian thực hiện của sắp xếp đống là $O(n \lg n)$. Giống như sắp xếp chèn, nhưng khác với sắp xếp trộn, kỹ thuật sắp xếp đống sắp xếp tại chỗ: vào một lúc bất kỳ chỉ một số mảng bất biến các thành phần được lưu trữ bên ngoài mảng đầu vào. Như vậy, sắp xếp đống tổ hợp các thuộc tính tốt hơn của hai thuật toán sắp xếp mà ta đã đề cập.

Sắp xếp đống cũng giới thiệu một kỹ thuật thiết kế thuật toán khác: vệt dụng một cấu trúc dữ liệu, trong trường hợp này ta gọi là một "đống" [heap], để quản lý thông tin trong khi thi hành thuật toán. Không những cấu trúc dữ liệu đống tỏ ra hữu ích với tiến trình sắp xếp đống, nó còn tạo một hàng đợi ưu tiên hiệu quả. Cấu trúc dữ liệu đống sẽ xuất hiện lại trong các thuật toán ở các chương sau.

Lưu ý, thoạt đầu thuật ngữ "đống" được đặt ra trong ngữ cảnh sắp xếp đống, nhưng từ đó nó lại được dùng để chỉ "khu lưu trữ thu gom rác," như trong ngôn ngữ lập trình Lisp. Cấu trúc dữ liệu đống của chúng ta không phải là khu lưu trữ thu gom rác, và mỗi khi nói về đống trong sách này, ta ám chỉ cấu trúc đã được định nghĩa trong chương này.

7.1 Đống

Cấu trúc dữ liệu ***đống (nhị phân)*** là một mảng đối tượng có thể được xem như một cây nhị phân hoàn chỉnh (xem Đoạn 5.5.3), như đã nêu trong Hình 7.1. Mỗi nút của cây tương ứng với một thành phần của mảng lưu trữ giá trị trong nút đó. Cây được điền đầy đủ trên tất cả các cấp ngoại trừ cấp thấp nhất khả dĩ, được điền từ trái lên cho đến một điểm. Một mảng A biểu thị cho một đống là một đối tượng có hai thuộc tính: $length[A]$, là số lượng thành phần trong mảng, và $heap\text{-}size[A]$, là số lượng thành phần trong đống lưu trữ trong mảng A. Nghĩa là, tuy $A[1..length[A]]$ có thể chứa các con số hợp lệ, song không có thành phần nào vượt quá $A[heap\text{-}size[A]]$, ở đó $heap\text{-}size[A] \leq length[A]$, là một thành phần của đống. Gốc của cây là $A[1]$, và căn cứ vào chỉ số i của một nút, các chỉ số của cha nó PARENT(i), con trái LEFT(i), và con phải RIGHT(i) có thể được tính toán là:



Hình 7.1 Một đống được xem là **(a)** một cây nhị phân và **(b)** một mảng. Con số bên trong vòng tròn tại mỗi nút trong cây là giá trị lưu trữ tại nút đó. Con số cạnh một nút là chỉ số tương ứng trong mảng.

PARENT(i)
    return $\lfloor i/2 \rfloor$
LEFT(i)
    return $2i$
RIGHT(i)
    return $2i + 1$

Trên hầu hết các máy tính, thủ tục LEFT có thể tính toán $2i$ trong một chỉ lệnh bằng cách đơn giản chuyển phần biểu thị nhị phân của i sang trái một vị trí bit. Cũng vậy, thủ tục RIGHT có thể nhanh chóng tính toán $2i + 1$ bằng cách chuyển phần biểu thị nhị phân của i sang trái một vị trí bit và chuyển vào một 1 làm bit cấp thấp. Thủ tục PARENT có thể tính toán $\lfloor i/2 \rfloor$ bằng cách chuyển i sang phải một vị trí bit. Trong một thực thi thích hợp của thuật toán sắp xếp đống, ba thủ tục này thường được thực thi dưới dạng các thủ tục "macro" hoặc "nội trình" [in-line].

Các đống cũng thỏa ***tính chất đống*** : với mọi nút i khác ngoài gốc,

$$A[\text{PARENT}(i)] \geq A[i]\,, \qquad (7.1)$$

nghĩa là, giá trị của một nút tối đa là giá trị của cha nó. Như vậy, thành phần lớn nhất trong một đống được lưu trữ tại gốc, và các cây con có gốc tại một nút chứa các giá trị nhỏ hơn chính nút đó.

Ta định nghĩa ***chiều cao*** [height] của một nút trong một cây là số lượng cạnh trên lộ trình đơn giản dài nhất đổ xuống từ nút đến một lá, và ta định nghĩa chiều cao của cây là chiều cao của gốc của nó. Bởi một đống n thành phần dựa trên một cây nhị phân hoàn chỉnh, nên chiều cao của nó là $\Theta(\lg n)$ (xem Bài tập 7.1-2). Ta sẽ thấy các phép toán cơ bản trên các đống chạy trong thời gian tỷ lệ tối đa với chiều cao của cây

và như vậy trải qua $O(\lg n)$ thời gian. Phần còn lại của chương này sẽ trình bày năm thủ tục cơ bản và cách dùng chúng trong một thuật toán sắp xếp và một cấu trúc dữ liệu hàng đợi ưu tiên.

- Thủ tục HEAPIFY, chạy trong $O(\lg n)$ thời gian, là chìa khóa để duy trì tính chất đống (7.1).
- Thủ tục BUILD-HEAP, chạy trong thời gian tuyến tính, tạo ra một đống từ một mảng đầu vào không sắp xếp.
- Thủ tục HEAPSORT, chạy trong $O(n \lg n)$ thời gian, sắp xếp một mảng tại chỗ.
- Thủ tục EXTRACT-MAX và INSERT, chạy trong $O(\lg n)$ thời gian, cho phép dùng cấu trúc dữ liệu đống làm hàng đợi ưu tiên.

Bài tập

7.1-1

Đâu là số lượng tối đa và tối thiểu các thành phần trong một đống có chiều cao h?

7.1-2

Chứng tỏ một đống n-thành phần có chiều cao $\lfloor \lg n \rfloor$.

7.1-3

Chứng tỏ thành phần lớn nhất trong một cây con của một đống nằm tại gốc của cây con.

7.1-4

Thành phần nhỏ nhất có thể thường trú tại đâu trong một đống?

7.1-5

Một mảng có phải là một đống [heap] được sắp xếp theo thứ tự nghịch đảo không?

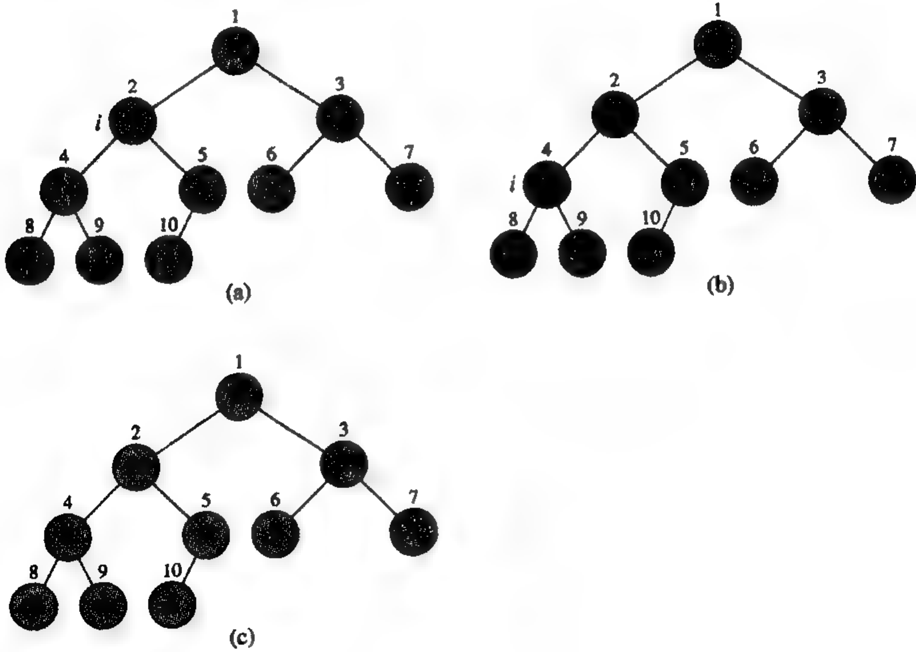
7.1-6

Dãy <23, 17, 14, 6, 13, 10, 1, 5, 7,12> có phải là một đống?

7.2 Duy trì tính chất đống

HEAPIFY là một chương trình con quan trọng để điều tác các đống. Đầu vào của nó là một mảng A và một chỉ số i vào mảng đó. Khi HEAPIFY được gọi, ta mặc nhận các cây nhị phân có gốc tại LEFT(i) và RIGHT(i) là các đống, nhưng $A[i]$ có thể nhỏ hơn các con của nó, như vậy sẽ vi phạm tính chất đống (7.1). Chức năng của HEAPIFY đó là để giá trị tại $A[i]$ "trôi xuống" [float down] trong đống sao cho cây

con có gốc tại chỉ số i trở thành một đống.



Hình 7.2 Hành động của HEAPIFY(A, 2), ở đó *heap-size*[A] = 10. **(a)** Cấu hình ban đầu của đống, với $A[2]$ tại nút $i = 2$ vi phạm tính chất đống bởi nó is not lớn hơn than cả hai các con. Tính chất đống được phục hồi cho nút 2 trong **(b)** bằng cách trao đổi $A[2]$ với $A[4]$, hủy tính chất đống của nút 4. Lệnh gọi đệ quy HEAPIFY(A, 4) giờ đây ấn định $i = 4$. Sau khi tráo $A[4]$ với $A[9]$, như đã nêu trong **(c)**, nút 4 được sửa lại, và lệnh gọi đệ quy HEAPiFY(A, 9) không cho ra thay đổi nào thêm đối với cấu trúc dữ liệu.

```
HEAPIFY(A, i)
    l ← LEFT(i)
    r ← RIGHT(i)
    if l ≤ heap-size[A] và A[l] > A[i]
        then largest ← l
        else largest ← i
    if r ≤ heap-size[A] và A[r] > A[largest]
        then largest ← r
    if largest ≠ i
        then exchange A[i] ↔ A[largest]
             HEAPIFY(A, largest)
```

Hình 7.2 minh họa hành động của HEAPIFY. Tại mỗi bước, thành

phần lớn nhất của các thành phần $A[i]$, $A[\text{LEFT}(i)]$, và $A[\text{RIGHT}(i)]$ sẽ được xác định, và chỉ số của nó được lưu trữ trong *largest*. Nếu $A[i]$ là lớn nhất, thì cây con có gốc tại nút i là một đống và thủ tục kết thúc. Bằng không, một trong hai con sẽ có thành phần lớn nhất, và $A[i]$ được tráo với $A[largest]$, khiến nút i và các con của nó thỏa tính chất đống. Tuy nhiên, nút *largest* giờ đây có giá trị ban đầu $A[i]$, và như vậy cây con có gốc tại *largets* có thể vi phạm tính chất đống. Bởi vậy, HEAPIFY phải được gọi đệ quy trên cây con đó.

Thời gian thực hiện của HEAPIFY trên một cây con có kích cỡ n có gốc tại nút đã cho i là $\Theta(1)$ thời gian để sửa lại các quan hệ giữa các thành phần $A[i]$, $A[\text{LEFT}(i)]$, và $A[\text{RIGHT}(i)]$, cộng với thời gian để chạy HEAPIFY trên một cây con có gốc tại một trong các con của nút i. Từng cây con của các con có kích cỡ tối đa $2n/3$—trường hợp xấu nhất xảy ra khi hàng cuối của cây chính xác đầy phân nửa—và như vậy thời gian thực hiện của HEAPIFY có thể được mô tả bằng phép truy toán

$$T(n) \leq T(2n/3) + \Theta(1) .$$

Nghiệm cho phép truy toán này, theo trường hợp 2 của định lý chủ (Định lý 4.1), là $T(n) = O(\lg n)$. Một cách khác, ta có thể định rõ đặc điểm thời gian thực hiện của HEAPIFY trên một nút có chiều cao h là $O(h)$.

Bài tập

7.2-1

Dùng Hình 7.2 như một chế độ, minh họa phép toán của HEAPIFY $(A, 3)$ trên mảng $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$.

7.2-2

Nêu hiệu ứng của việc gọi HEAPIFY(A, i) khi thành phần $A[i]$ lớn hơn các con của nó?

7.2-3

Nêu hiệu ứng của việc gọi HEAPIFY(A, i) với $i > heap\text{-}size[A]/2$?

7.2-4

Mã của HEAPIFY khá hiệu quả theo dạng các thừa số bất biến, ngoại trừ có thể với lệnh gọi đệ quy trong dòng 10, nó có thể khiến vài bộ biên dịch tạo ra mã không hiệu quả. Viết một HEAPIFY hiệu quả sử dụng một cấu trúc điều khiển lặp (một vòng lặp) thay vì đệ quy.

7.2-5

Chứng tỏ thời gian thực hiện ca xấu nhất của HEAPIFY trên một đống có kích cỡ n là $\Omega(\lg n)$. (*Mách nước.* Với một đống có n nút, nêu các giá trị nút khiến HEAPIFY được gọi đệ quy tại mọi nút trên một lộ trình từ gốc đổ xuống một lá.)

7.3 Xây dựng một đống

Có thể dùng thủ tục HEAPIFY theo cách dưới-lên để chuyển đổi một mảng $A[1..n]$, ở đó $n = length[A]$ thành một đống. Bởi các thành phần trong mảng con $A[(\lfloor n/2 \rfloor + 1)..n]$ là tất cả các lá của cây, nên có thể bắt đầu bằng mỗi thành phần là một đống 1-thành phần. Thủ tục BUILD-HEAP đi qua các nút còn lại của cây và chạy HEAPIFY trên từng nút một. Thứ tự qua đó các nút được xử lý sẽ bảo đảm các cây con có gốc tại các con của một nút i là các đống trước khi chạy HEAPIFY tại nút đó.

```
BUILD-HEAP(A)
heap-size[A] ← length[A]
for i ← ⌊length[A]/2⌋ downto 1
        do HEAPIFY(A, i)
```

Hình 7.3 có nêu một ví dụ về hành động của BUILD-HEAP.

Có thể tính toán một cận trên đơn giản trên thời gian thực hiện của BUILD-HEAP như sau. Mỗi lệnh gọi đến HEAPIFY hao phí $O(\lg n)$ thời gian, và có $O(n)$ lệnh gọi như vậy. Như vậy, thời gian thực hiện tối đa là $O(n \lg n)$. Tuy là đúng, song cận trên này không sát theo tiệm cận.

Có thể suy ra một cận sát hơn bằng cách nhận xét rằng thời gian để HEAPIFY chạy tại một nút sẽ thay đổi theo chiều cao của nút đó trong cây, và chiều cao của hầu hết các nút đều là nhỏ. Tiến trình phân tích chặt hơn của chúng ta dựa vào tính chất mà trong một đống n-thành phần có tối đa $\lceil n/2^{h+1} \rceil$ nút có chiều cao h (xem Bài tập 7.3-3).

Thời gian mà HEAPIFY yêu cầu khi được gọi trên một nút có chiều cao h là $O(h)$, do đó ta có thể diễn tả tổng hao phí của BUILD-HEAP là

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) . \tag{7.2}$$

Phép lấy tổng cuối có thể được đánh giá bằng cách thay $x = 1/2$ trong công thức (3.6), cho ra

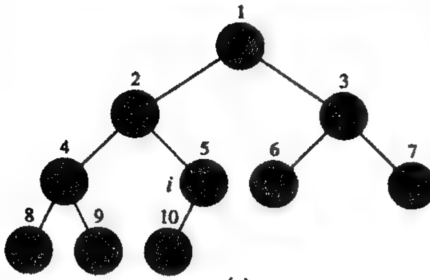
$$\begin{aligned} \sum_{h=0}^{\infty} \frac{h}{2^h} &= \frac{1/2}{(1 - 1/2)^2} \\ &= 2 . \end{aligned}$$

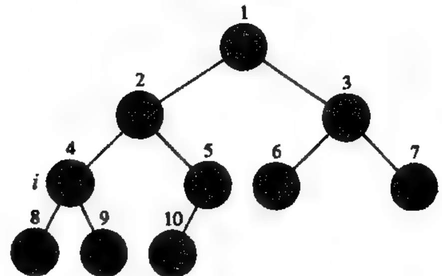
Như vậy, thời gian thực hiện của BUILD-HEAP có thể được định cận là

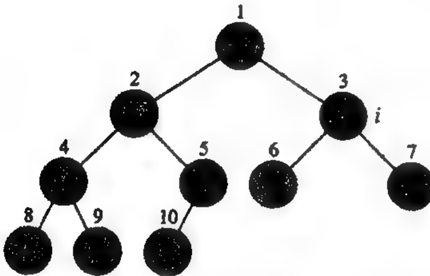
$$O\left(n\sum_{h=0}^{\lfloor \lg n\rfloor}\frac{h}{2^h}\right) = O\left(n\sum_{h=0}^{\infty}\frac{h}{2^h}\right)$$
$$= O(n)\ .$$

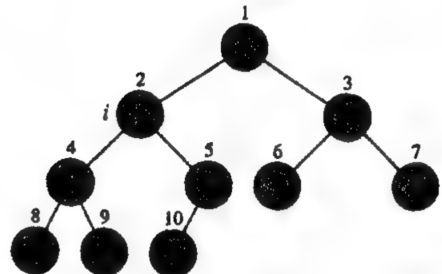
Do đó, ta có thể xây dựng một đống từ một mảng không sắp xếp trong thời gian tuyến tính.

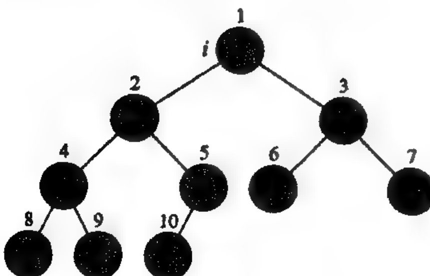
A [redacted]

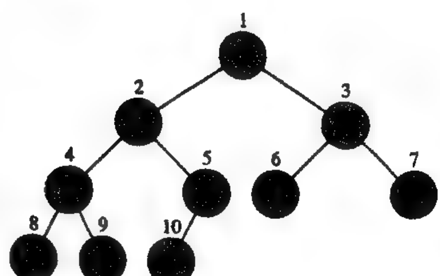












Hình 7.3 Phép toán của BUILD-HEAP, nêu cấu trúc dữ liệu trước lệnh gọi đến HEAPIFY trong dòng 3 của BUILD-HEAP. **(a)** Một mảng đầu vào *A* 10-thành phần và cây nhị phân mà nó biểu thị. Hình cho thấy chỉ số vòng lặp *i* trỏ đến nút 5 trước lệnh gọi HEAPIFY(*A*, *i*). **(b)** Cấu trúc dữ liệu kết quả. Chỉ số vòng lặp *i* với lần lặp lại kế tiếp trỏ đến nút 4. **(c)-(e)** Các lần lặp tiếp theo của vòng lặp **for** trong BUILD-HEAP. Nhận thấy mỗi khi HEAPIFY được gọi trên một nút, cả hai cây con của nút đó đều là các đống. **(f)** Đống sau khi BUILD-HEAP hoàn tất.

Bài tập

7.3-1

Dùng Hình 7.3 làm mô hình, minh họa phép toán của BUILD-HEAP trên mảng A = <5, 3, 17,10, 84, 19, 6, 22, 9>.

7.3-2

Tại sao ta lại muốn chỉ số vòng lặp i trong dòng 2 của BUILD-HEAP giảm từ $\lfloor length[A]/2 \rfloor$ đến 1 thay vì tăng từ 1 đến $\lfloor length[A]/2 \rfloor$?

7.3-3

Chứng tỏ có tối đa $\lceil n/2^{h+1} \rceil$ nút có chiều cao h trong bất kỳ đống n-thành phần nào.

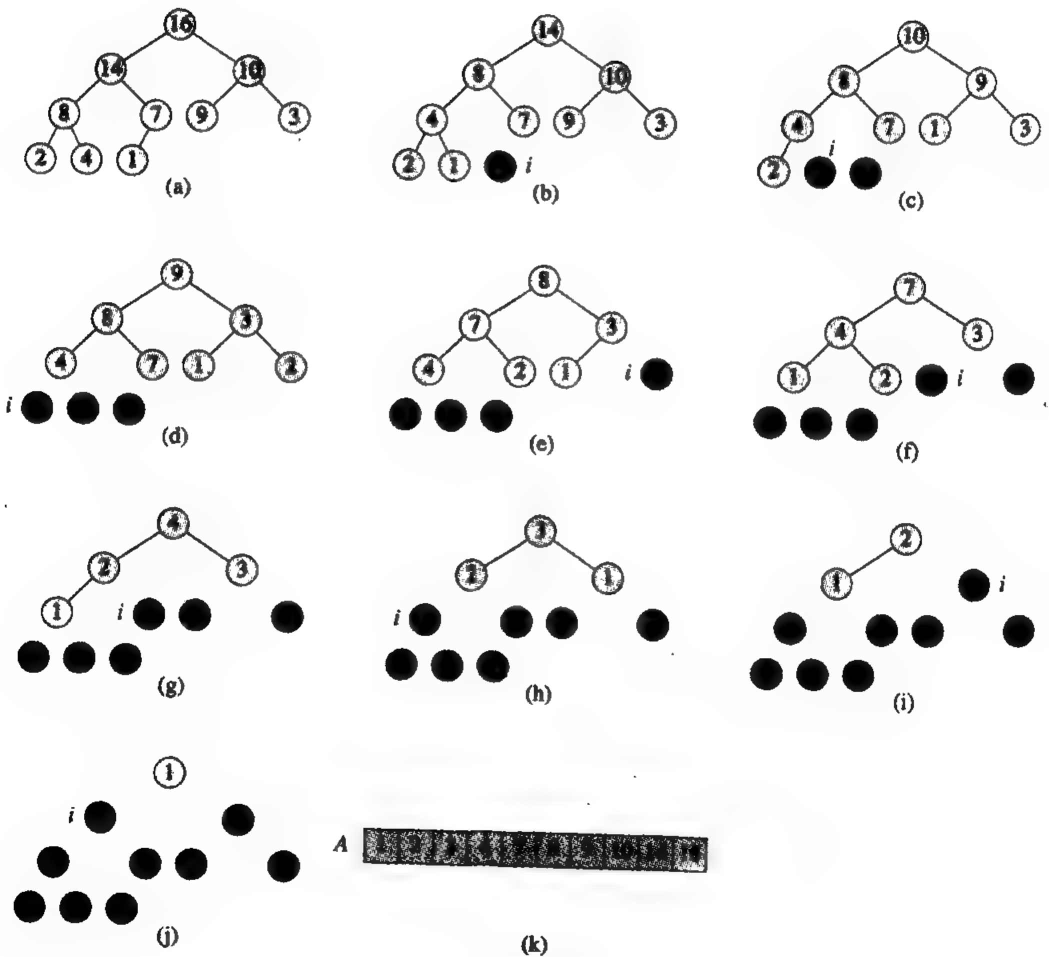
7.4 Thuật toán sắp xếp đống

Thuật toán sắp xếp đống bắt đầu bằng cách dùng BUILD-HEAP để xây dựng một đống trên mảng đầu vào $A[1..n]$, ở đó $n = length[A]$. Bởi thành phần cực đại của mảng được lưu trữ tại gốc $A[1]$, nên ta có thể đặt nó vào đúng vị trí chung cuộc của nó bằng cách trao đổi nó với $A[n]$. Nếu giờ đây ta "thải bỏ" nút n ra khỏi đống (bằng cách giảm số *heap-size*[A]), ta nhận thấy $A[1..(n-1)]$ có thể dễ dàng được tạo thành một đống. Các con của gốc giữ lại các đống, nhưng thành phần gốc mới có thể vi phạm tính chất đống (7.1). Tuy nhiên, để phục hồi tính chất đống, ta chỉ cần một lệnh gọi đến HEAPIFY(A, 1), để một đống trong $A[1..(n-1)]$. Sau đó thuật toán sắp xếp đống lặp lại tiến trình này với đống có kích cỡ n - 1 đổ xuống đến một đống có kích cỡ 2.

```
HEAPSORT(A)
1 BUILD-HEAP(A)
2 for i ← length[A] downto 2
3       do exchange A[1] ↔ A[i]
4          heap-size[A] ← heap-size[A] - 1
5          HEAPIFY(A, 1)
```

Hình 7.4 có nêu một ví dụ về phép toán sắp xếp đống sau khi đống được xây dựng ban đầu. Mỗi đống được nêu tại đầu một lần lặp lại của vòng lặp **for** trong dòng 2.

Thủ tục HEAPSORT mất một thời gian $O(n \lg n)$, bởi lệnh gọi đến BUILD-HEAP mất một thời gian $O(n)$ và mỗi trong số các lệnh gọi n - 1 đến HEAPIFY mất một thời gian $O(\lg n)$.



Hình 7.4 Phép toán của HEAPSORT. **(a)** Cấu trúc dữ liệu đống ngay sau khi nó đã được xây dựng bởi BUILD-HEAP. **(b)-(j)** Đống ngay sau mỗi lệnh gọi của HEAPIFY trong dòng 5. Giá trị của *i* vào lúc đó được nêu. Chỉ các nút được tô bóng nhẹ sẽ giữ lại trong đống. **(k)** Mảng kết quả được sắp xếp *A*.

Bài tập

7.4-1

Dùng Hình 7.4 làm mô hình, minh họa phép toán của HEAPSORT trên mảng A = <5, 13, 2, 25, 7, 17, 20, 8, 4>.

7.4-2

Đâu là thời gian thực hiện của thuật toán sắp xếp đống trên một mảng A có chiều dài n được sắp xếp sẵn theo thứ tự tăng? Còn về thứ tự giảm thì sao?

7.4-3

Chứng tỏ thời gian thực hiện của sắp xếp đống là $\Omega(n \lg n)$.

7.5 Các hàng đợi ưu tiên

Sắp xếp đống là một thuật toán tuyệt vời, nhưng trong thực tế, một thực thi tốt của sắp xếp nhanh [quicksort], xem Chương 8, thường đánh bại nó. Tuy vậy, bản thân cấu trúc dữ liệu đống có tính tiện ích thật to lớn. Đoạn này trình bày một trong các ứng dụng phổ dụng nhất của một đống: dùng nó như một hàng đợi ưu tiên hiệu quả.

Một ***hàng đợi ưu tiên*** [priority queue] là một cấu trúc dữ liệu để duy trì một tập hợp S các thành phần, mỗi thành phần có một giá trị kết hợp có tên là ***khóa*** [key]. Một hàng đợi ưu tiên hỗ trợ các phép toán dưới đây.

INSERT(S, x) chèn thành phần x vào tập hợp S. Phép toán này có thể được viết dưới dạng $S \leftarrow S \cup \{x\}$.

MAXIMUM(S) trả về thành phần của S có khóa lớn nhất.

EXTRACT-MAX(S) gỡ bỏ và trả về thành phần của S có khóa lớn nhất.

Một ứng dụng của các hàng đợi ưu tiên đó là lên lịch các khối xử lý [jobs] trên một máy tính dùng chung. Hàng đợi ưu tiên sẽ theo dõi các khối xử lý sẽ được thực hiện và các mức ưu tiên tương đối của chúng. Khi một khối xử lý hoàn tất hoặc bị ngắt, khối xử lý có mức ưu tiên cao nhất sẽ được lựa trong số các khối xử lý đang chờ giải quyết bằng EXTRACT-MAX. Một khối xử lý mới có thể được bổ sung vào hàng đợi bất kỳ lúc nào bằng INSERT.

Một hàng đợi ưu tiên cũng có thể được dùng trong một bộ mô phỏng điều khiển theo sự kiện. Các mục trong hàng đợi là các sự kiện được mô phỏng, mỗi mục có một thời gian xảy ra kết hợp được dùng làm

khóa. Các sự kiện phải được mô phỏng theo thứ tự thời gian xảy ra của chúng, bởi vì việc mô phỏng một sự kiện có thể khiến các sự kiện khác được mô phỏng trong tương lai. Với ứng dụng này, quả tự nhiên khi đảo ngược thứ tự tuyến tính của hàng đợi ưu tiên và hỗ trợ các phép toán MINIMUM và EXTRACT-MIN, thay vì MAXIMUM và EXTRACT-MAX. Chương trình mô phỏng sử dụng EXTRACT-MIN tại mỗi bước để chọn sự kiện kế tiếp sẽ mô phỏng. Khi các sự kiện mới được tạo, chúng được chèn vào hàng đợi ưu tiên bằng INSERT.

Tất nhiên, ta có thể dùng một đống để thực thi một hàng đợi ưu tiên. Phép toán HEAP-MAXIMUM trả về thành phần đống cực đại trong $\Theta(1)$ thời gian bằng cách chỉ việc trả về giá trị $A[1]$ trong đống. Thủ tục HEAP-EXTRACT-MAX cũng tương tự như thân vòng lặp **for** (các dòng 3-5) của thủ tục HEAPSORT:

```
HEAP-EXTRACT-MAX(A)
if heap-size[A] < 1
   then error "heap underflow"
max ← A[1]
A[1] ← A[heap-size[A]]
heap-size[A] ← heap-size[A] - 1
HEAPIFY(A, 1)
return max
```

Thời gian thực hiện của HEAP-EXTRACT-MAX là $O(\lg n)$, bởi nó chỉ thực hiện một lượng công việc bất biến trên đầu $O(\lg n)$ thời gian với HEAPIFY.

Thủ tục HEAP-INSERT chèn một nút vào đống *A*. Để làm thế, trước tiên nó khai triển đống bằng cách bổ sung một lá mới vào cây. Sau đó, giống như vòng lặp chèn (các dòng 5-7) của INSERTION-SORT từ Đoạn 1.1, nó ngang qua một lộ trình từ lá này về phía gốc để tìm ra một vị trí thích hợp cho thành phần mới.

```
HEAP-INSERT(A, key)
heap-size[A] ← heap-size[A] + 1
i ← heap-size[A]
while i > 1 and A[PARENT(i)] < key
    do A[i] ← A[PARENT(i)]
       i ← PARENT(i)
A[i] ← key
```

Hình 7.5 có nêu một ví dụ của một phép toán HEAP-INSERT. Thời gian thực hiện của HEAP-INSERT trên một đống n-thành phần là $O(\lg n)$, bởi lộ trình lần theo từ lá mới đến gốc có chiều dài $O(\lg n)$.

Tóm lại, một đống có thể hỗ trợ bất kỳ phép toán hàng đợi ưu tiên nào trên một tập hợp có kích cỡ n trong $O(\lg n)$ thời gian.

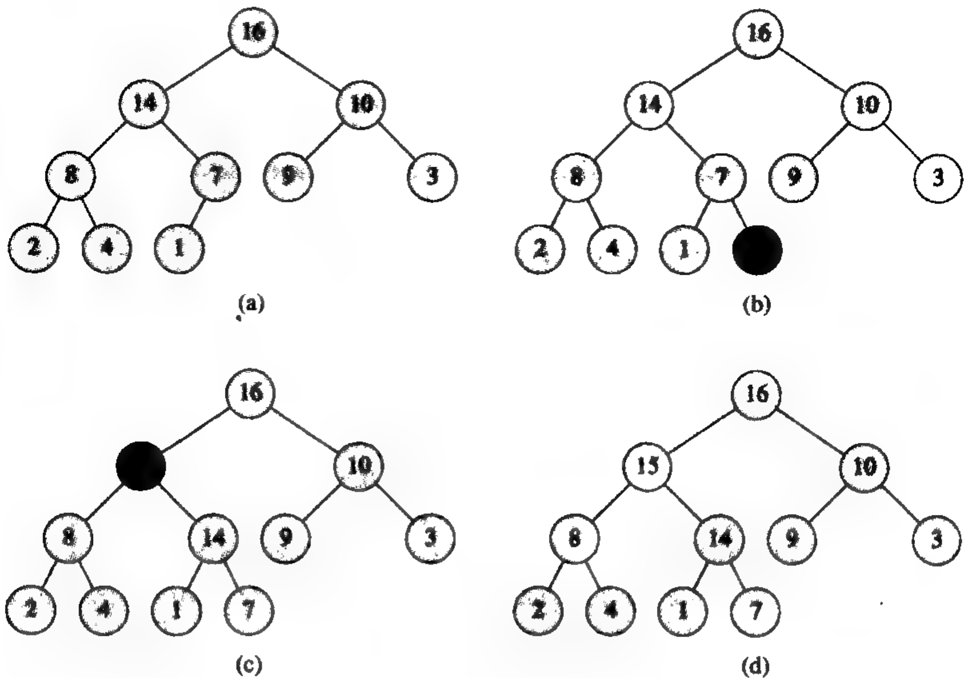
Bài tập

7.5-1

Dùng Hình 7.5 làm mô hình, minh họa phép toán của HEAP-INSERT(A, 3) trên đống A = <15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1>.

7.5-2

Minh họa phép toán của HEAP-EXTRACT-MAX trên đống A = <15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2,1>.



Hình 7.5 Phép toán của HEAP-INSERT. **(a)** Đống của Hình 7.4(a) trước khi chèn một nút có khóa 15. **(b)** Một lá mới được bổ sung vào cây. **(c)** Các giá trị trên lộ trình từ lá mới đến gốc được chép xuống cho đến khi tìm thấy một nơi cho khóa 15. **(d)** Khóa 15 được chèn.

7.5-3

Nêu cách thực thi hàng đợi vào trước ra trước có một hàng đợi ưu tiên. Nêu cách thực thi một ngăn xếp với một hàng đợi ưu tiên. (Đoạn

11.1 có định nghĩa FIFO và các ngăn xếp.)

7.5-4

Nêu một thực thi $O(\lg n)$-thời gian của thủ tục HEAP-INCREASEKEY(A, i, k), cái nào sẽ ấn định $A[i] \leftarrow \max(A[i], k)$ và cập nhật cấu trúc đống thích hợp.

7.5-5

Phép toán HEAP-DELETE(A, i) xóa mục trong nút i ra khỏi đống A. Nêu một thực thi của HEAP-DELETE chạy trong $O(\lg n)$ thời gian với một đống n-thành phần.

7.5-6

Nêu một thuật toán $O(n \lg k)$-thời gian để trộn k danh sách đã sắp xếp vào một danh sách đã sắp xếp, ở đó n là tổng của các thành phần trong tất cả các danh sách đầu vào. (*Mách nước:* Dùng một đống cho tiến trình trộn k-cách.)

Các Bài Toán

7-1 Xây dựng một đống bằng kỹ thuật chèn

Thủ tục BUILD-HEAP trong Đoạn 7.3 có thể được thực thi bằng cách dùng lặp HEAP-INSERT để chèn các thành phần vào đống. Xem xét kiểu thực thi dưới đây:

```
BUILD-HEAP' (A)
heap-size[A] ← 1
for i ← 2 to length[A]
      do HEAP-INSERT(A, A[i])
```

a. Các thủ tục BUILD-HEAP và BUILD-HEAP' có luôn tạo cùng đống khi chạy trên cùng mảng đầu vào không? Chứng minh chúng có, hoặc cung cấp một ví dụ phản lại.

b. Chứng tỏ trong ca xấu nhất, BUILD-HEAT' yêu cầu $\Theta(n \lg n)$ thời gian để xây dựng một đống n-thành phần.

7-2 Phân tích các đống thuộc-d

Một ***đống thuộc-d*** [d-ary heap] giống như một đống nhị phân, nhưng thay vì 2 con, các nút có d con.

a. Làm sao để biểu thị một đống thuộc-d trong một mảng?

b. Cho biết chiều cao của một đống thuộc-d gồm n thành phần theo dạng n và d?

c. Nêu một kiểu thực thi khác của EXTRACT-MAX. Phân tích thời gian thực hiện của nó theo dạng d và n.

d. Nêu một kiểu thực thi hiệu quả của INSERT. Phân tích thời gian thực hiện của nó theo dạng d và n.

e. Nêu một kiểu thực thi hiệu quả của HEAP-INCREASING-KEY(A, i, k), ấn định $A[i] \leftarrow \max(A[i], k)$ và cập nhật cấu trúc đống thích hợp. Phân tích thời gian thực hiện của nó theo dạng d và n.

Ghi chú Chương

Thuật toán sắp xếp đống được Williams [202] phát minh, ông cũng là người đã mô tả cách thực thi một hàng đợi ưu tiên có một đống. Thủ tục BUILD-HEAP được Floyd [69] gợi ý.

8 Sắp Xếp Nhanh

Sắp xếp nhanh là một thuật toán sắp xếp có thời gian thực hiện trường hợp xấu nhất là $\Theta(n^2)$ trên một mảng đầu vào n con số. Mặc dù thời gian thực hiện trường hợp xấu nhất của nó chậm, sắp xếp nhanh vẫn là chọn lựa thực tiễn tốt nhất để sắp xếp bởi xét bình quân nó mang tính hiệu quả đáng nể: thời gian thực hiện dự trù của nó là $\Theta(n \lg n)$, và các thừa số bất biến ẩn trong hệ ký hiệu $\Theta(n \lg n)$ là khá nhỏ. Nó cũng có ưu điểm về sắp xếp tại chỗ (xem trang 8), và nó làm việc tốt kể cả trong các môi trường bộ nhớ ảo.

Đoạn 8.1 mô tả thuật toán và một chương trình con quan trọng được kỹ thuật sắp xếp nhanh sử dụng để phân hoạch. Bởi cách ứng xử của sắp xếp nhanh phức hợp, nên ta bắt đầu bằng việc giải thích trực giác khả năng thực hiện của nó trong Đoạn 8.2 và chừa phần phân tích chính xác vào cuối chương. Đoạn 8.3 trình bày hai phiên bản của sắp xếp nhanh dùng bộ phát sinh ngẫu số giả. Các thuật toán "ngẫu nhiên hóa" này có nhiều tính chất thỏa đáng. Thời gian thực hiện trường hợp trung bình là tốt, và không có đầu vào cụ thể nào suy ra cách ứng xử trường hợp xấu nhất của chúng. Một trong các phiên bản ngẫu nhiên hóa của sắp xếp nhanh được phân tích trong Đoạn 8.4, ở đó nó được nêu để chạy trong $O(n^2)$ thời gian trong trường hợp xấu nhất và trong $O(n \lg n)$ thời gian trong trường hợp trung bình.

8.1 Mô tả kiểu sắp xếp nhanh

Cũng như sắp xếp trộn, sắp xếp nhanh dựa trên kiểu mẫu chia để trị đã nêu trong Đoạn 1.3.1. Dưới đây là tiến trình chia để trị ba-bước khi sắp xếp một mảng con điển hình $A[p..r]$.

Chia: Mảng $A[p..r]$ được phân hoạch (dàn xếp lại) thành hai mảng con không trống $A[p..q]$ và $A[q + 1..r]$ sao cho mỗi thành phần của $A[p..q]$ nhỏ hơn hoặc bằng với từng thành phần của $A[q + 1..r]$. Chỉ số q được tính toán như một phần của thủ tục phân hoạch này.

Trị: Hai mảng con $A[p..q]$ và $A[q + 1..r]$ được sắp xếp bằng các lệnh gọi đệ quy để sắp xếp nhanh.

Tổ hợp: Bởi các mảng con được sắp xếp tại chỗ, nên không cần công sức để tổ hợp chúng: nguyên cả mảng $A[p..r]$ giờ đây được sắp xếp.

Thủ tục dưới đây thực thi thuật toán sắp xếp nhanh.

```
QUICKSORT(A,p, r)
if p < r
   then q ← PARTITION(A, p, r)
        QUICKSORT(A,p,q)
        QUICKSORT(A, q + 1, r)
```

Để sắp xếp nguyên cả mảng A, lệnh gọi ban đầu là QUICKSORT (A,1, *length*[A]).

Phân hoạch mảng

Trọng tâm của thuật toán là thủ tục PARTITION, nó dàn xếp lại mảng con $A[p..r]$ tại chỗ.

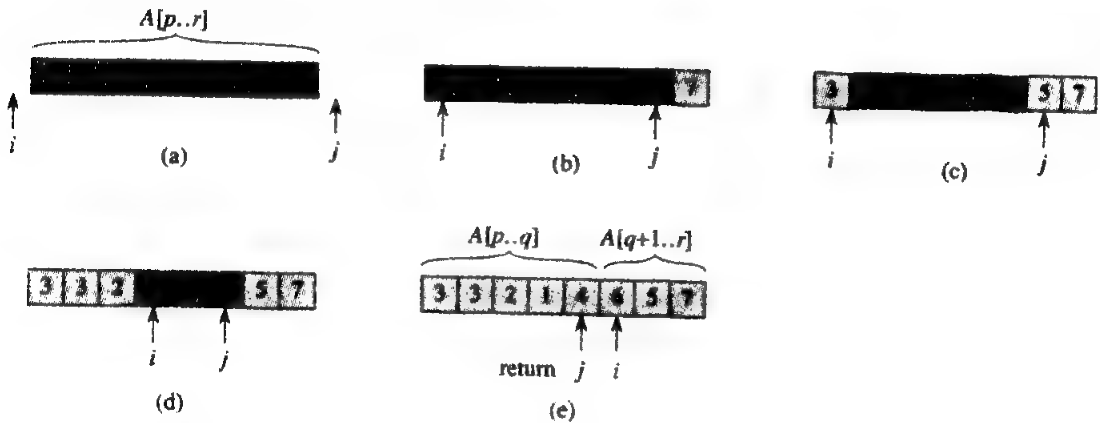
```
PARTITION(A, p, r)
x ← A[p]
i ← p-1
j ← r + 1
while TRUE
      do repeat j ← j - 1
            until A[j] ≤ x
         repeat i ← i + 1
            until A[i] ≥ x
         if i < j
            then exchange A[i] ↔ A[j]
            else return j
```

Hình 8.1 có nêu cách làm việc của PARTITION. Trước tiên nó lựa một thành phần $x = A[p]$ từ $A[p..r]$ làm thành phần "trục" [pivot] để phân hoạch $A[p..r]$ quanh nó. Sau đó tăng trưởng hai vùng $A[p..i]$ và $A[j..r]$ từ đỉnh và đáy của $A[p..r]$, theo thứ tự nêu trên, sao cho mọi thành phần trong $A[p..i]$ nhỏ hơn hoặc bằng với x và mọi thành phần trong $A[j..r]$ lớn hơn hoặc bằng với x. Thoạt đầu, $i = p - 1$ và $j = r + 1$, do đó hai vùng là trống.

Bên trong thân của vòng lặp **while**, chỉ số j được giảm số

[decremented] và chỉ số i được gia số, trong các dòng 5-8, cho đến khi $A[i] \geq x \geq A[j]$. Giả sử các bất đẳng thức này là nghiêm ngặt, $A[i]$ quá lớn không thể thuộc về vùng đáy và $A[j]$ quá nhỏ không thể thuộc về vùng đỉnh. Như vậy, qua trao đổi $A[i]$ và $A[j]$ như trong dòng 10, ta có thể mở rộng hai vùng. (Nếu các bất đẳng thức không nghiêm ngặt, việc trao đổi có thể được tiến hành tùy ý.)

Thân của vòng lặp **while** lặp lại cho đến khi $i \geq j$, tại điểm này nguyên cả mảng $A[p..r]$ đã được phân hoạch thành hai mảng con $A[p..q]$ và $A[q+1..r]$, ở đó $p \leq q < r$, sao cho không có thành phần nào của $A[p..q]$ lớn hơn bất kỳ thành phần nào của $A[q+1..r]$. Giá trị $q = j$ được trả về tại cuối thủ tục.



Hình 8.1 Phép toán của PARTITION trên một mảng mẫu. Các thành phần mảng tô bóng sáng đã được đặt vào đúng phân hoạch, và các thành phần tô bóng đậm chưa nằm trong các phân hoạch của chúng. **(a)** Mảng đầu vào, có các giá trị ban đầu i và j vuột ra khỏi các đầu trái và phải của mảng. Ta phân hoạch quanh $x = A[p] = 5$. **(b)** Các vị trí của i và j tại dòng 9 của lần lặp đầu tiên của vòng lặp **while**. **(c)** Kết quả của việc trao đổi các thành phần được i và j trỏ đến trong dòng 10. **(d)** Các vị trí của i và j tại dòng 9 của lần lặp thứ hai của vòng lặp **while**. **(e)** Các vị trí của i và j tại dòng 9 của lần lặp lại thứ ba và chót của vòng lặp **while**. Thủ tục kết thúc bởi vì $i \geq j$, và giá trị $q = j$ được trả về. Các thành phần mảng lên đến và kể cả $A[j]$ sẽ nhỏ hơn hoặc bằng với $x = 5$, và các thành phần mảng sau $A[j]$ sẽ lớn hơn hoặc bằng với $x = 5$.

Về khái niệm, thủ tục phân hoạch thực hiện một chức năng đơn giản: nó đặt các thành phần nhỏ hơn x vào vùng đáy của mảng và các thành phần lớn hơn x vào vùng đỉnh. Tuy nhiên, có các chi tiết kỹ thuật hơi tinh tế khi tạo mã giả của PARTITION. Ví dụ, các chỉ số i và j không bao giờ lập chỉ mục mảng con $A[p..r]$ vượt ngoài các cận, nhưng điều

này không hoàn toàn hiển nhiên qua mã. Một ví dụ khác, điều quan trọng đó là $A[p]$ được dùng như thành phần trục [pivot] x. Nếu thay vào đó $A[r]$ được dùng và tình cờ $A[r]$ cũng là thành phần lớn nhất trong mảng con $A[p..r]$, thì PARTITION trả về cho QUICKSORT giá trị $q = r$, và QUICKSORT sẽ lặp vòng mãi. Bài toán 8-1 yêu cầu bạn chứng minh PARTITION là đúng.

Thời gian thực hiện của PARTITION trên một mảng $A[p..r]$ is $\Theta(n)$, ở đó $n = r - p + 1$ (xem Bài tập 8.1-3).

Bài tập

8.1-1

Dùng Hình 8.1 làm mô hình, minh họa phép toán của PARTITION trên mảng $A = <13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21>$.

8.1-2

PARTITION sẽ trả về giá trị q nào khi tất cả các thành phần trong mảng $A[p..r]$ có cùng giá trị?

8.1-3

Biện luận ngắn gọn thời gian thực hiện của PARTITION trên mảng con có kích cỡ n là $\Theta(n)$.

8.1-4

Sửa đổi QUICKSORT như thế nào để sắp xếp theo thứ tự không tăng?

8.2 Khả năng thực hiện của sắp xếp nhanh

Thời gian thực hiện của thuật toán sắp xếp nhanh tùy thuộc vào việc phân hoạch có cân đối hay không, và đến lượt điều này lại tùy thuộc vào các thành phần được dùng để phân hoạch. Nếu tiến trình phân hoạch là cân đối, thì theo tiệm cận thuật toán chạy nhanh như sắp xếp trộn. Tuy nhiên, nếu tiến trình phân hoạch không cân đối, thì theo tiệm cận nó có thể chạy chậm như sắp xếp chèn. Trong đoạn này, ta sẽ nghiên cứu qua cách thực hiện của thuật toán sắp xếp nhanh dưới các giả thiết phân hoạch cân đối và không cân đối.

Phân hoạch theo trường hợp xấu nhất

Cách ứng xử trong trường hợp xấu nhất của sắp xếp nhanh xảy ra khi thường trình phân hoạch tạo ra một vùng với $n - 1$ thành phần và một vùng với chỉ 1 thành phần. (Tuyên bố này được chứng minh trong Đoạn

8.4.1.) Ta mặc nhận rằng tiến trình phân hoạch không cân đối này nảy sinh tại mọi bước của thuật toán. Bởi tiến trình phân hoạch hao phí $\Theta(n)$ thời gian và $T(1) = \Theta(1)$, phép truy toán của thời gian thực hiện là

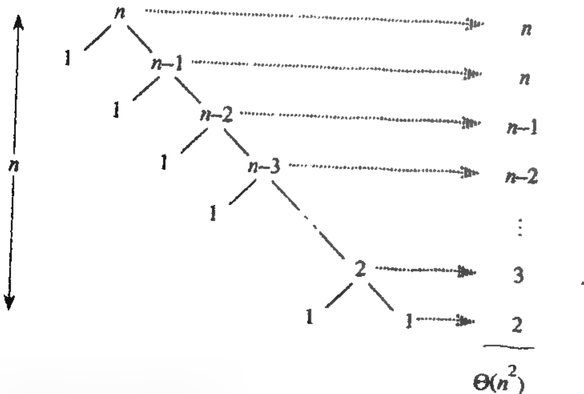
$$T(n) = T(n-1) + \Theta(n).$$

Để đánh giá phép truy toán này, ta nhận thấy $T(1) = \Theta(1)$ rồi lặp lại:

$$\begin{aligned} T(n) &= T(n-1) + \Theta(n) \\ &= \sum_{k=1}^{n} \Theta(k) \\ &= \Theta\left(\sum_{k=1}^{n} k\right) \\ &= \Theta(n^2). \end{aligned}$$

Ta được dòng cuối bằng cách nhận xét rằng $\sum_{k=1}^{n} k$ là chuỗi cấp số cộng (3.2). Hình 8.2 có nêu một cây đệ quy của tiến trình thi hành trường hợp xấu nhất của thuật toán sắp xếp nhanh. (Xem Đoạn 4.2 để biết thêm về các cây đệ quy.)

Như vậy, nếu tiến trình phân hoạch không cân đối tối đa tại mọi bước đệ quy của thuật toán, thời gian thực hiện là $\Theta(n^2)$. Do đó thời gian thực hiện trường hợp xấu nhất của sắp xếp nhanh không tốt hơn thuật toán sắp xếp chèn. Hơn nữa, thời gian thực hiện $\Theta(n^2)$ xảy ra khi mảng đầu vào đã được sắp xếp sẵn hoàn toàn—một tình huống chung ở đó sắp xếp chèn chạy trong $O(n)$ thời gian.



Hình 8.2 Một cây đệ quy của QUICKSORT ở đó thủ tục PARTITION luôn đặt độc một thành phần đơn lẻ trên một phía của phân hoạch (trường hợp xấu nhất). Thời gian thực hiện kết quả là $\Theta(n^2)$.

Tiến trình phân hoạch theo trường hợp tốt nhất

Nếu thủ tục phân hoạch tạo ra hai vùng có kích cỡ $n/2$, thuật toán sắp xếp nhanh sẽ chạy nhanh hơn nhiều. Vậy phép truy toán là

$T(n) = 2T(n/2) + \Theta(n)$,

Mà theo trường hợp 2 của định lý chủ (Định lý 4.1) có nghiệm $T(n) = \Theta(n \lg n)$. Như vậy, tiến trình phân hoạch trường hợp tốt nhất này sẽ tạo ra một thuật toán nhanh hơn nhiều. Hình 8.3 có nêu cây đệ quy của tiến trình thi hành trường hợp tốt nhất của thuật toán sắp xếp nhanh.

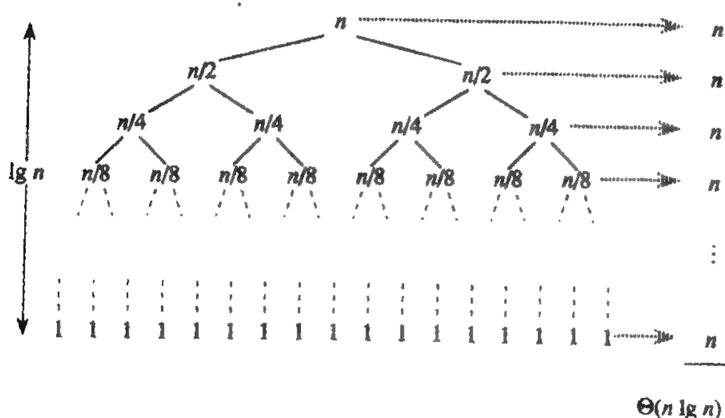
Tiến trình phân hoạch cân đối

Thời gian thực hiện trường hợp trung bình của sắp xếp nhanh gần với trường hợp tốt nhất nhiều hơn so với trường hợp xấu nhất, như các đợt phân tích trong Đoạn 8.4 cho thấy. Mấu chốt để hiểu tại sao điều này có thể là đúng đó là tìm hiểu cách phản ánh sự cân đối của tiến trình phân hoạch trong phép truy toán mô tả thời gian thực hiện.

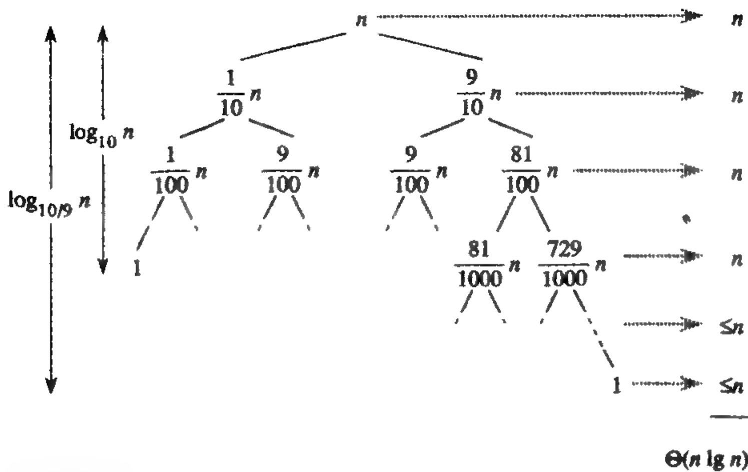
Ví dụ, giả sử thuật toán phân hoạch luôn tạo ra một đợt tách theo tỷ lệ 9-1, mà thoạt nhìn dường như chẳng mấy cân đối. Vậy ta được phép truy toán

$$T(n) = T(9n/10) + T(n/10) + n$$

trên thời gian thực hiện của sắp xếp nhanh, ở đó để tiện dụng ta đã thay $\Theta(n)$ bằng n. Hình 8.4 có nêu cây đệ quy của phép truy toán này. Lưu ý, mọi cấp của cây có hao phí n, cho đến khi đạt được một điều kiện cận tại chiều sâu $\log_{10} n = \Theta(\lg n)$, sau đó các cấp có hao phí tối đa n. Phép đệ quy kết thúc tại chiều sâu $\log_{10/9} n = \Theta(\lg n)$. Do đó, tổng hao phí của thuật toán sắp xếp nhanh là $\Theta(n \lg n)$. Như vậy, với một đợt tách theo tỷ lệ 9-1 tại mọi cấp của đệ quy, mà theo trực giác dường như không mấy cân đối, thuật toán sắp xếp nhanh chạy trong $\Theta(n \lg n)$ thời gian—mà theo tiệm cận cũng giống như thể đợt tách đã xuống nằm ngay giữa. Thực vậy, kể cả một đợt tách 99-1 cũng cho ra $O(n \lg n)$ thời



Hình 8.3 Một cây đệ quy của QUICKSORT ở đó PARTITION luôn cân đối hai phía của phân hoạch bằng nhau (trường hợp tốt nhất). Thời gian thực hiện kết quả là $\Theta(n \lg n)$.



Hình 8.4 Một cây đệ quy của QUICKSORT ở đó PARTITION luôn tạo ra một đợt tách 9-1, cho ra một thời gian thực hiện $\Theta(n \lg n)$.

gian thực hiện. Lý do đó là bất kỳ đợt tách nào có tính tỷ lệ *bất biến* đều cho ra một cây đệ quy có chiều sâu $\Theta(\lg n)$, ở đó hao phí tại mỗi cấp là $O(n)$. Do đó thời gian thực hiện là $\Theta(n \lg n)$ mỗi khi đợt tách có tính tỷ lệ bất biến.

Trực giác đối với trường hợp trung bình

Để phát triển một khái niệm rõ rệt về ca trung bình cho thuật toán sắp xếp nhanh, ta phải tạo một giả thiết về mức độ thường xuyên dự trù sẽ gặp các đầu vào khác nhau. Một giả thiết chung đó là tất cả các phép hoán vị của các con số đầu vào đều có khả năng như nhau. Ta sẽ giải thích giả thiết này trong đoạn sau, nhưng trước nút ta khảo sát các nhánh của nó.

Khi chạy thuật toán sắp xếp nhanh trên một mảng đầu vào ngẫu nhiên, không chắc tiến trình phân hoạch luôn xảy ra giống nhau tại mọi cấp, như tiến trình phân tích không chính thức của chúng ta đã mặc nhận. Ta dự trù một số đợt tách sẽ cân đối hợp lý và một số sẽ không cân đối mấy. Ví dụ, Bài tập 8.2-5 yêu cầu bạn chứng tỏ khoảng 80 phần trăm thời gian mà PARTITION tạo ra một đợt tách cân đối hơn so với 9-1, và khoảng 20 phần trăm thời gian mà nó tạo ra một đợt tách ít cân đối hơn so với 9-1.

Trong trường hợp trung bình, PARTITION tạo ra một hỗn hợp các đợt tách "tốt" và "xấu". Trong một cây đệ quy dánh cho tiến trình thi hành trường hợp trung bình của PARTITION, các đợt tách tốt và xấu được phân phối ngẫu nhiên xuyên suốt cây. Tuy nhiên, vì trực giác, giả sử các đợt tách tốt và xấu làm thay đổi các cấp trong cây, và các đợt tách tốt là trường hợp tốt nhất và các đợt tách xấu là trường hợp xấu

nhất. Hình 8.5(a) có nêu các đợt tách tại hai cấp liên tiếp trong cây đệ quy. Tại gốc của cây, hao phí là n để phân hoạch và các mảng con được tạo sẽ có các kích cỡ $n - 1$ và 1: trường hợp xấu nhất. Tại cấp kế tiếp, mảng con có kích cỡ n-1 được phân hoạch theo trường hợp tốt nhất thành hai mảng con có kích cỡ $(n - 1)/2$. Giả sử hao phí điều kiện cận là 1 với mảng con có kích cỡ 1.

Tổ hợp đợt tách xấu theo sau là đợt tách tốt tạo ra ba mảng con có các kích cỡ 1, $(n - 1)/2$, và $(n - 1)/2$ tại một mức hao phí phối hợp $2n - 1 = \Theta(n)$. Tất nhiên, tình huống này không tệ hơn trong Hình 8.5(b), tức là một cấp đơn lẻ của tiến trình phân hoạch tạo ra hai mảng con có kích cỡ $(n - 1)/2 + 1$ và $(n - 1)/2$ với một mức hao phí $n = \Theta(n)$. Hơn nữa tình huống sau hầu như cân đối, chắc chắn tốt hơn 9-1. Theo trực giác, hao phí $\Theta(n)$ của đợt tách xấu có thể được hấp thụ vào hao phí $\Theta(n)$ của đợt tách tốt, và đợt tách kết quả là tốt. Như vậy, thời gian thực hiện của thuật toán sắp xếp nhanh, khi các cấp thay đổi giữa các đợt tách tốt và xấu, giống như thời gian thực hiện của chỉ mình các đợt tách tốt: vẫn $O(n \lg n)$, nhưng với một hằng hơi lớn hơn được hệ ký hiệu O che giấu. Ta sẽ phân tích chính xác trường hợp trung bình trong Đoạn 8.4.2.



Hình 8.5 **(a)** Hai cấp của một cây đệ quy dành cho thuật toán sắp xếp nhanh. Tiến trình phân hoạch tại gốc hao phí n và tạo ra một đợt tách "xấu": hai mảng con có các kích cỡ 1 và $n - 1$. Tiến trình phân hoạch của mảng con có kích cỡ $n - 1$ sẽ hao phí $n - 1$ và tạo ra một đợt tách "tốt": hai mảng con có kích cỡ $(n - 1)/2$. **(b)** Một cấp đơn lẻ của một cây đệ quy tệ hơn các cấp phối hợp trong (a), song lại rất cân đối.

Bài tập

8.2-1

Chứng tỏ thời gian thực hiện của QUICKSORT là $\Theta(n \lg n)$ khi tất cả các thành phần của mảng A có cùng giá trị.

8.2-2

Chứng tỏ thời gian thực hiện của QUICKSORT là $\Theta(n^2)$ khi mảng A được lưu trữ theo thứ tự không tăng.

8.2-3

Các ngân hàng thường bút toán các giao dịch trên một tài khoản theo thứ tự thời gian của các giao dịch, nhưng nhiều người thích nhận các bản khai tình hình ngân hàng của họ với các chi phiếu liệt kê theo thứ tự số chi phiếu. Ta thường viết các chi phiếu theo thứ tự số chi phiếu, và các thương gia thường có thể nhanh chóng đổi chúng thành tiền mặt. Như vậy, bài toán chuyển đổi kiểu sắp xếp thứ tự theo thời gian giao dịch thành kiểu sắp xếp theo số chi phiếu là bài toán sắp xếp đầu vào hầu như đã được sắp xếp sẵn. Chứng tỏ thủ tục INSERTION-SORT có khuynh hướng đánh bại thủ tục QUICKSORT trong bài toán này.

8.2-4

Giả sử các đợt tách tại mọi cấp sắp xếp nhanh theo tỷ lệ $1 - \alpha$ đối α, ở đó $0 < \alpha \leq 1/2$ là một hằng. Chứng tỏ chiều sâu tối thiểu của một lá trong cây đệ quy xấp xỉ $- \lg n / lg\alpha$ và chiều sâu tối đa xấp xỉ $- \lg n / \lg(1 - \alpha)$. (Đừng bận tâm về việc làm tròn số nguyên.)

*8.2-5 **

Biện luận với bất kỳ hằng $0 < \alpha \leq 1/2$, xác suất xấp xỉ $1 - 2\alpha$ mà trên một mảng đầu vào ngẫu nhiên, PARTITION tạo ra một đợt tách cân đối hơn $1 - \alpha$ đối α. Với giá trị α nào sẽ là các số lẻ cho dù đợt tách cân đối hơn ít cân đối?

8.3 Các phiên bản ngẫu nhiên hóa của sắp xếp nhanh

Để khảo sát cách ứng xử trường hợp trung bình của thuật toán sắp xếp nhanh, ta có lập một giả thiết rằng tất cả các phép hoán vị của các con số đầu vào có khả năng như nhau. Khi giả thiết này trên phép phân phối các đầu vào là hợp lệ, nhiều người xem kỹ thuật sắp xếp nhanh như một thuật toán chọn lựa đối với các đầu vào đủ lớn. Tuy nhiên, trong tình huống thiết kế kỹ thuật, ta không thể luôn kỳ vọng áp dụng nó. (Xem Bài tập 8.2-3.) Đoạn này giới thiệu khái niệm của một thuật toán ngẫu nhiên hóa và trình bày hai phiên bản ngẫu nhiên hóa của sắp xếp nhanh khắc phục giả thiết cho rằng tất cả các phép hoán vị của các con số đầu vào có khả năng như nhau.

Một cách khác thay cho *việc mặc nhận* một phép phân phối các đầu vào đó là *áp đặt* một phép phân phối. Ví dụ, giả sử trước khi sắp xếp mảng đầu vào, thuật toán sắp xếp nhanh ngẫu nhiên hoán vị các thành phần để củng cố tính chất mà mọi phép hoán vị có khả năng như nhau. (Bài tập 8.3-4 yêu cầu một thuật toán ngẫu nhiên hoán vị các thành phần của một mảng có kích cỡ n trong thời gian $O(n)$.) Việc sửa đổi này

không cải thiện thời gian thực hiện trường hợp xấu nhất của thuật toán, nhưng nó khiến thời gian thực hiện độc lập với tiến trình sắp xếp đầu vào.

Ta gọi một thuật toán là ***ngẫu nhiên hóa*** [randomized] nếu cách ứng xử của nó được xác định không những bởi đầu vào mà còn bởi các giá trị do một ***bộ phát sinh ngẫu số*** tạo ra. Mặc nhận rằng ta có thể tùy ý sử dụng một bộ phát sinh ngẫu số RANDOM. Một lệnh gọi đến RANDOM(a, b) trả về một số nguyên giữa a và b, kể cả hai giá trị này, với từng số nguyên như vậy có khả năng như nhau. Ví dụ, RANDOM(0, 1) tạo ra một 0 với xác suất 1/2 và một 1 với xác suất 1/2. Mỗi số nguyên được RANDOM trả về sẽ độc lập với các số nguyên được trả về trên các lệnh gọi trước đó. Bạn có thể tưởng tượng RANDOM như đang lắc một con xúc xắc có (b - a + 1) cạnh để được kết xuất của nó. (Trong thực tế, hầu hết môi trường lập trình đều cung cấp một ***bộ phát sinh ngẫu số giả***: một thuật toán tất định trả về các con số "có vẻ" ngẫu nhiên theo thống kê.)

Phiên bản ngẫu nhiên hóa này của thuật toán sắp xếp nhanh có một tính chất thú vị mà nhiều thuật toán ngẫu nhiên hóa khác cũng có: *không có đầu vào cụ thể nào gợi ra cách ứng xử trường hợp xấu nhất của nó.* Thay vì thế, trường hợp xấu nhất của nó tùy thuộc vào bộ phát sinh ngẫu số. Kể cả khi cố ý, bạn không thể tạo ra một mảng đầu vào xấu cho sắp xếp nhanh, bởi phép hoán vị ngẫu nhiên khiến thứ tự đầu vào trở thành không thích hợp. Thuật toán ngẫu nhiên hóa chỉ thực hiện xấu nêu như bộ phát sinh ngẫu số tạo ra một phép hoán vị không may mắn sẽ được sắp xếp. Bài tập 13.4-4 chứng tỏ hầu như tất cả các phép hoán vị đều khiến thuật toán sắp xếp nhanh thực hiện gần tốt như trường hợp trung bình: có *rất ít* phép hoán vị gây ra cách ứng xử trường hợp gần xấu nhất.

Chiến lược ngẫu nhiên hóa thường hữu ích khi có nhiều cách có thể tiến hành một thuật toán song lại khó xác định một cách bảo đảm là tốt. Nếu có nhiều cách là tốt, việc đơn giản chọn một theo ngẫu nhiên có thể dẫn đến một chiến lược tốt. Thông thường, một thuật toán phải thực hiện nhiều chọn lựa trong khi thi hành. Nếu các lợi ích của các chọn lựa tốt vượt trên các hao phí của các chọn lựa xấu, việc lựa chọn ngẫu nhiên các chọn lựa tốt và xấu có thể cho ra một thuật toán hiệu quả. Ta đã lưu ý trong Đoạn 8.2, một hỗn hợp các đợt tách tốt và xấu cho ra một thời gian thực hiện tốt đối với thuật toán sắp xếp nhanh, và như vậy quả có ý nghĩa khi các phiên bản ngẫu nhiên hóa của thuật toán thực hiện tốt.

Nhờ sửa đổi thủ tục PARTITION, ta có thể thiết kế một phiên bản ngẫu nhiên hóa khác của thuật toán sắp xếp nhanh sử dụng chiến lược

chọn lựa ngẫu nhiên này. Tại mỗi bước của thuật toán sắp xếp nhanh, trước khi mảng được phân hoạch, ta trao đổi thành phần $A[p]$ với một thành phần chọn ngẫu nhiên từ $A[p..r]$. Kiểu sửa đổi này bảo đảm thành phần trục [pivot] $x = A[p]$ có khả năng ngang nhau là bất kỳ trong số $r - p + 1$ thành phần trong mảng con. Như vậy, ta kỳ vọng đợt tách của mảng đầu vào theo bình quân sẽ khá cân đối. Thuật toán ngẫu nhiên hóa dựa vào việc hoán vị ngẫu nhiên mảng đầu vào cũng làm việc tốt theo trung bình, nhưng nó hơi khó phân tích hơn so với phiên bản này.

Các thay đổi đối với PARTITION và QUICKSORT không lớn. Trong thủ tục phân hoạch mới, ta chỉ việc thực thi việc tráo đổi trước khi tiến hành phân hoạch thực tế:

```
RANDOMIZED-PARTITION(A, p, r)
  i ← RANDOM(p, r)
  exchange A[p] ↔ A[i]
  return PARTITION(A, p, r)
```

Giờ đây ta tạo một thuật toán sắp xếp nhanh mới tên RANDOMIZED-PARTITION thay vì PARTITION:

```
RANDOMIZED-QUICKSORT(A, p, r)
  if p < r
      then q ← RANDOMIZED-PARTITION(A, p, r)
           RANDOMIZED-QUICKSORT(A, p, q)
           RANDOMIZED-QUICKSORT(A, q + 1, r)
```

Ta sẽ phân tích thuật toán này trong đoạn sau.

Bài tập

8.3-1

Tại sao ta phân tích khả năng thực hiện trường hợp trung bình của một thuật toán ngẫu nhiên hóa mà không phải khả năng thực hiện trường hợp xấu nhất của nó?

8.3-2

Trong khi chạy thủ tục RANDOMIZED-QUICKSORT, bao nhiêu lệnh gọi được thực hiện đến bộ phát sinh ngẫu số RANDOM trong trường hợp xấu nhất? Câu trả lời sẽ thay đổi ra sao trong trường hợp tốt nhất?

8.3-3 *

Mô tả một thực thi của thủ tục RANDOM(a, b) chỉ sử dụng các lần

lật đồng tiền cân. Đâu là thời gian thực hiện dự trù của thủ tục?

8.3-4 *

Nêu một thủ tục ngẫu nhiên hóa, $\Theta(n)$ - thời gian lấy một mảng $[1..n]$ làm đầu vào và thực hiện một phép hoán vị ngẫu nhiên trên các thành phần mảng.

8.4 Phân tích thuật toán sắp xếp nhanh

Đoạn 8.2 đã đưa ra vài trực giác về cách ứng xử trường hợp xấu nhất của sắp xếp nhanh và về việc tại sao ta dự trù nó chạy nhanh. Đoạn này sẽ phân tích cách ứng xử của thuật toán sắp xếp nhanh chính xác hơn. Ta bắt đầu bằng việc phân tích trường hợp xấu nhất, áp dụng cho QUICKSORT hoặc RANDOMIZED-QUICKSORT, và kết thúc bằng một phân tích trường hợp trung bình với RANDOMIZED-QUICKSORT.

8.4.1 Phân tích trường hợp xấu nhất

Trong Đoạn 8.2 ta đã thấy một đợt tách trường hợp xấu nhất tại mọi cấp đệ quy trong sắp xếp nhanh sẽ tạo ra một thời gian thực hiện $\Theta(n^2)$, mà, theo trực giác, là thời gian thực hiện trường hợp xấu nhất của thuật toán. Giờ đây ta chứng minh sự quyết đoán này.

Dùng phương pháp thay thế (xem Đoạn 4.1), ta có thể chứng tỏ thời gian thực hiện của thuật toán sắp xếp nhanh là $O(n^2)$. Cho $T(n)$ là thời gian trường hợp xấu nhất đối với thủ tục QUICKSORT trên một đầu vào có kích cỡ n. Ta có phép truy toán

$$T(n) = \max_{1 \le q \le n-1} (T(q) + T(n-q)) + \Theta(n), \qquad (8.1)$$

ở đó tham số q nằm trong phạm vi từ 1 đến n - 1 bởi vì thủ tục PARTITION tạo ra hai vùng, mỗi vùng có kích cỡ ít nhất là 1. Ta suy đoán $T(n) \le cn^2$ với một hằng c nào đó. Thay suy đoán này vào (8.1), ta được

$$\begin{aligned} T(n) &\le \max_{1 \le q \le n-1} (cq^2 + c(n-q)^2) + \Theta(n) \\ &= c \cdot \max_{1 \le q \le n-1} (q^2 + (n-q)^2) + \Theta(n). \end{aligned}$$

Biểu thức $q^2 + (n-q)^2$ đạt được một cực đại trên miền giá trị $1 \le q \le n-1$ tại một trong các điểm cuối, như có thể thấy bởi đạo hàm thứ hai của biểu thức theo q là dương (xem Bài tập 8.4-2). Điều này cho ta cận $\max_{1 \le q \le n-1}(q^2 + (n-q)^2) \le 1^2 + (n-1)^2 = n^2 - 2(n-1)$.

Tiếp tục với cách định cận $T(n)$, ta được

$$\begin{aligned} T(n) &\le cn^2 - 2c(n-1) + \Theta(n) \\ &\le cn^2, \end{aligned}$$

bởi ta có thể chọn hằng c đủ lớn sao cho số hạng $2c(n - 1)$ chi phối số hạng $\Theta(n)$. Như vậy, thời gian thực hiện (trường hợp xấu nhất) của sắp xếp nhanh là $\Theta(n^2)$.

8.4.2 Phân tích trường hợp trung bình

Ta đã biện luận theo trực giác tại sao thời gian thực hiện ca trung bình của RANDOMIZED-QUICKSORT lại là $\Theta(n \lg n)$: nếu đợt tách được cảm sinh bởi RANDOMIZED-PARTITION đặt bất kỳ phân số bất biến nào của các thành phần trên một phía của phân hoạch, thì cây đệ quy sẽ có chiều sâu $\Theta(\lg n)$ và $\Theta(n)$ công việc được thực hiện tại $\Theta(\lg n)$ của các cấp này. Để có thể phân tích thời gian thực hiện dự trù của RANDOMIZED-QUICKSORT một cách chính xác, trước tiên ta phải hiểu rõ cách hoạt động của thủ tục phân hoạch. Sau đó, ta có thể phát triển một phép truy toán đối với thời gian trung bình cần thiết để sắp xếp một mảng n-thành phần và giải phép truy toán này để xác định các cận trên thời gian thực hiện dự trù. Với tư cách là một phần của tiến trình giải phép truy toán, ta sẽ phát triển các cận sát [tight bounds] trên một phép lấy tổng đang lưu ý.

Phân tích tiến trình phân hoạch

Trước tiên có vài nhận xét về phép toán của PARTITION. Khi PARTITION được gọi trong dòng 3 của thủ tục RANDOMIZED-PARTITION, thành phần $A[p]$ đã được trao đổi với một thành phần ngẫu nhiên trong $A[p..r]$. Để đơn giản hóa tiến trình phân tích, ta mặc nhận rằng tất cả các con số nhập đều riêng biệt. Nếu tất cả các con số nhập không riêng biệt, thời gian thực hiện trường hợp trung bình của sắp xếp nhanh vẫn đúng là $O(n \lg n)$, nhưng cần phải có một phân tích hơi phức tạp hơn so với phân tích được trình bày ở đây.

Nhận xét đầu tiên là: giá trị q mà PARTITION trả về chỉ tùy thuộc vào hạng của $x = A[p]$ giữa các thành phần trong $A[p..r]$. (***Hạng*** [rank] của một con số trong một tập hợp là số lượng thành phần nhỏ hơn hoặc bằng với nó.) Nếu cho $n = r - p + 1$ là số lượng thành phần trong $A[p..r]$, việc tráo đổi $A[p]$ với một thành phần ngẫu nhiên từ $A[p..r]$ cho ra một xác suất $1/n$ mà $\text{rank}(x) = i$ với $i = 1, 2, ..., n$.

Kế đó ta tính toán các khả năng khả dĩ của các kết quả khác nhau trong tiến trình phân hoạch. Nếu $\text{rank}(x) = 1$, thì lần đầu tiên qua vòng lặp **while** trong các dòng 4-11 của PARTITION, chỉ số i dừng tại $i = p$ và chỉ số j dừng tại $j = p$. Như vậy, khi $q = j$ được trả về, phía "thấp" của phân hoạch chứa thành phần độc nhất $A[p]$. Sự kiện này xảy ra với xác suất $1/n$ bởi đó là xác suất mà $\text{rank}(x) = 1$.

Nếu $\text{rank}(x) \geq 2$, thì có ít nhất một thành phần nhỏ hơn $x = A[p]$. Bởi

vậy, lần đầu tiên qua vòng lặp **while**, chỉ số i dừng tại $i = p$ nhưng j dừng trước khi đạt đến p. Như vậy một trao đổi với $A[p]$ được thực hiện để đặt $A[p]$ trong phía cao của phân hoạch. Khi PARTITION kết thúc, mỗi trong số rank(x) - 1 thành phần trong phía thấp của phân hoạch hoàn toàn nhỏ hơn x. Như vậy, với mỗi $i = 1, 2,..., n - 1$, khi rank(x) ≥ 2, xác suất là $1/n$ mà phía thấp của phân hoạch có i thành phần.

Phối hợp hai trường hợp này, ta kết luận rằng kích cỡ $q - p + 1$ của phía thấp của phân hoạch là 1 với xác suất $2/n$ và kích cỡ là i với xác suất $1/n$ với $i = 2, 3,..., n - 1$.

Phép truy toán trong trường hợp trung bình

Giờ đây ta thiết lập một phép truy toán cho thời gian thực hiện dự trù của RANDOMIZED-QUICKSORT. Cho $T(n)$ thể hiện thời gian trung bình cần thiết để sắp xếp một mảng đầu vào n-thành phần. Một lệnh gọi đến RANDOMIZED-QUICKSORT với một mảng 1-thành phần sẽ bỏ ra một thời gian bất biến, do đó ta có $T(1) = \Theta(1)$. Một lệnh gọi đến RANDOMIZED-QUICKSORT với một mảng $A[1..n]$ có chiều dài n sử dụng $\Theta(n)$ thời gian để phân hoạch mảng. Thủ tục PARTITION trả về một chỉ số q, rồi RANDOMIZED-QUICKSORT được gọi theo đệ quy với các mảng con có chiều dài q và $n - q$. Bởi vậy, thời gian trung bình để sắp xếp một mảng có chiều dài n có thể được diễn tả là

$$T(n) = \frac{1}{n}\left(T(1) + T(n-1) + \sum_{q=1}^{n-1} (T(q) + T(n-q))\right) + \Theta(n) . \quad (8.2)$$

Giá trị q có một phép phân phối hầu như đồng đều, ngoại trừ giá trị $q = 1$ gần gấp đôi so với các giá trị khác, như đã lưu ý trên đây. Dùng sự việc $T(1) = \Theta(1)$ và $T(n - 1) = O(n^2)$ từ tiến trình phân tích trường hợp xấu nhất, ta có

$$\begin{aligned} \frac{1}{n}(T(1) + T(n-1)) &= \frac{1}{n}(\Theta(1) + O(n^2)) \\ &= O(n) , \end{aligned}$$

và do đó số hạng $\Theta(n)$ trong phương trình (8.2) có thể hấp thụ biểu thức $\frac{1}{n}(T(1) + T(n-1))$. Như vậy, ta có thể phát biểu lại phép truy toán (8.2) là

$$T(n) = \frac{1}{n}\sum_{q=1}^{n-1} (T(q) + T(n-q)) + \Theta(n) . \quad (8.3)$$

Nhận thấy với $k = 1, 2,..., n - 1$, mỗi term $T(k)$ của tổng xảy ra một lần dưới dạng $T(q)$ và một lần dưới dạng $T(n - q)$. Rút gọn hai số hạng của tổng sẽ cho ra

$$T(n) = \frac{2}{n}\sum_{k=1}^{n-1}(T(k) + \Theta(n)\,. \qquad (8.4)$$

Giải phép truy toán

Ta có thể giải phép truy toán (8.4) bằng phương pháp thay thế. Theo quy nạp, ta mặc nhận $T(n) \le a\,n \lg n + b$ với vài hằng $a > 0$ và $b > 0$ sẽ được xác định. Ta có thể chọn a và b đủ lớn sao cho $a \lg n + b$ lớn hơn $T(1)$. Vậy với $n > 1$, và bằng thay thế ta có

$$\begin{aligned} T(n) &= \frac{2}{n}\sum_{k=1}^{n-1}(T(k) + \Theta(n)\,. \\ &\le \frac{2}{n}\sum_{k=1}^{n-1}(ak\lg k + b) + \Theta(n) \\ &= \frac{2a}{n}\sum_{k=1}^{n-1}k\lg k + \frac{2b}{n}(n-1) + \Theta(n)\,. \end{aligned}$$

Dưới đây ta chứng tỏ phép lấy tổng trong dòng cuối có thể được định cận bằng

$$\sum_{k=1}^{n-1}k\lg k \le \frac{1}{2}n^2\lg n - \frac{1}{8}n^2 \qquad (8.5)$$

Dùng cận này, ta được

$$\begin{aligned} T(n) &\le \frac{2a}{n}\left(\frac{1}{2}n^2\lg n - \frac{2a}{n}n^2\right) + \frac{2b}{n}(n-1) + \Theta(n) \\ &\le an\lg n - \frac{a}{4}n + 2b + \Theta(n) \\ &= an\lg n + b + \left(\Theta(n) + b - \frac{a}{4}n\right) \\ &\le an\lg n + b\,, \end{aligned}$$

bởi ta có thể chọn a đủ lớn sao cho $\frac{a}{4}n$ chi phối $\Theta(n) + b$. Ta kết luận rằng thời gian thực hiện trung bình của sắp xếp nhanh là $O(n \lg n)$.

Các cận sát trên phép lấy tổng chính

Còn lại, ta chứng minh cận (8.5) trên phép lấy tổng

$$\sum_{k=1}^{n-1}k\lg k\,.$$

Bởi mỗi số hạng tối đa là $n \lg n$, ta có cận

$$\sum_{k=1}^{n-1}k\lg k \le n^2\lg n$$

sát trong một thừa số bất biến. Tuy nhiên, cận này không đủ mạnh để giải phép truy toán dưới dạng $T(n) = O(n \lg n)$. Cụ thể, ta cần một cận

của $\frac{1}{2} n^2 \lg n - \Omega(n^2)$ để tính nghiệm của phép truy toán.

Để tìm cận này trên phép lấy tổng, ta tách nó thành hai phần, như mô tả trong Đoạn 3.2. Ta được

$$\sum_{k=1}^{n-1} k \lg k = \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \lg k .$$

lg k trong phép lấy tổng đầu tiên bên phải được định cận bên trên bằng $\lg(n/2) = \lg n - 1$. lg k trong phép lấy tổng thứ hai được định cận bên trên bằng lg n. Như vậy,

$$\begin{aligned}
\sum_{k=1}^{n-1} k \lg k &= (\lg n - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k \\
&= \lg n \sum_{k=1}^{n-1} \sum_{k=1}^{\lceil n/2 \rceil - 1} k \\
&\le \frac{1}{2} n(n-1) \lg n - \frac{1}{2} \left(\frac{n}{2} - 1 \right) \frac{n}{2} \\
&\le \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2
\end{aligned}$$

nếu $n \ge 2$. Đây là cận (8.5).

Bài tập

8.4-1

Chứng tỏ thời gian thực hiện ca tốt nhất của sắp xếp nhanh là $\Omega(n \lg n)$.

8.4-2

Chứng tỏ $q^2 + (n - q)^2$ đạt đến một cực đại trên $q = 1, 2, \ldots, n - 1$ khi $q = 1$ hoặc $q = n-1$.

8.4-3

Chứng tỏ thời gian thực hiện dự trù của RANDOMIZED-QUICKSORT là $\Omega(n \lg n)$.

8.4-4

Trong thực tế, để cải thiện thời gian thực hiện của sắp xếp nhanh, ta vệt dụng thời gian thực hiện nhanh của thuật toán sắp xếp chèn khi đầu vào của nó "gần như" được sắp xếp. Khi thuật toán sắp xếp nhanh được gọi trên một mảng con có ít hơn k thành phần, hãy để nó đơn giản trả về mà không sắp xếp mảng con. Sau khi lệnh gọi cấp trên cùng đến thuật toán sắp xếp nhanh trở về, ta chạy thuật toán sắp xếp chèn trên nguyên cả mảng để hoàn tất tiến trình sắp xếp. Biện luận thuật toán sắp xếp này chạy trong thời gian dự trù $O(nk + n \lg(n/k))$. k sẽ được chọn như thế

nào, cả trong lý thuyết lẫn trong thực tế?

8.4-5 *

Chứng minh đồng nhất thức

$$\int x \ln x \, dx = \frac{1}{2} x^2 \ln x - \frac{1}{4} x^2 ,$$

rồi dùng phương pháp tích phân xấp xỉ để nêu một cận trên sát hơn (8.5) trên phép lấy tổng $\sum_{k=1}^{n-1} k \lg k$.

8.4-6 *

Xem xét việc sửa đổi thủ tục PARTITION bằng cách chọn ngẫu nhiên ba thành phần từ mảng *A* và phân hoạch khoảng trung tuyến của chúng. Tính xấp xỉ xác suất để có một đợt tách α - đối - $(1 - \alpha)$ theo trường hợp xấu nhất, dưới dạng một hàm của *a* trong miền giá trị $0 < \alpha < 1$.

Các Bài Toán

8-1 Tính đúng đắn của phân hoạch

Biện luận cẩn thận thủ tục PARTITION trong Đoạn 8.1 là đúng đắn. Chứng minh:

a. Các chỉ số *i* và *j* không bao giờ tham chiếu một thành phần của *A* bên ngoài khoảng $[p..r]$.

b. Chỉ số *j* không bằng *r* khi PARTITION kết thúc (sao cho đợt tách luôn không tầm thường).

c. Mọi thành phần của $A[p..j]$ nhỏ hơn hoặc bằng với mọi thành phần của $A[j+1..r]$ khi PARTITION kết thúc.

8-2 Thuật toán phân hoạch Lomuto

Xem xét biến thể của PARTITION dưới đây, do N. Lomuto. Để phân hoạch $A[p..r]$, phiên bản này tăng trưởng hai vùng, $A[p..i]$ và $A[i+1..j]$, sao cho mọi thành phần trong vùng đầu tiên nhỏ hơn hoặc bằng với $x = A[r]$ và mọi thành phần trong vùng thứ hai lớn hơn *x*.

```
LOMUTO-PARTITION (A, p, r)
x ← A[r]
i ← p - 1
for j ← p to r
    do if A[j] ≤ x
        then i ← i + 1
```

```
                    exchange A[i] ↔ A[j]
if i < r
    then return i
    else return i - 1
```

a. Chứng tỏ LOMUTO-PARTITION là đúng.

b. Nêu các số lần tối đa mà PARTITION và LOMUTO-PARTITION có thể dời một thành phần?

c. Biện luận LOMUTO-PARTITION, cũng như PARTITION, chạy trong $\Theta(n)$ thời gian trên một mảng con n-thành phần.

d. Việc thay PARTITION bằng LOMUTO-PARTITION có ảnh hưởng như thế nào đến thời gian thực hiện của QUICKSORT khi tất cả các giá trị nhập đều bằng nhau?

e. Định nghĩa một thủ tục RANDOMIZED-LOMUTO-PARTITION trao đổi $A[r]$ với một thành phần được chọn ngẫu nhiên trong $A[p..r]$ rồi gọi LOMUTOPARTITION. Chứng tỏ xác suất mà RANDOMIZED-LOMUTO-PARTITION trả về một giá trị đã cho ngang bằng với xác suất RANDOMIZED-PARTITION trả về $p + r - q$.

8-3 Sắp xếp Stooge

Các giáo sư Howard, Fine, và Howard đã đề xuất thuật toán sắp xếp "lịch lãm" dưới đây:

```
STOOGE-SORT(A, i, j)
if A[i] > A[j]
    then exchange A[i] ↔ A[j]
if i + 1 ≥ j
    then return
k ← ⌊(j - i + 1)/3⌋          ▷ Làm tròn xuống.
STOOGE-SORT(A, i, j - k)     ▷ Hai phần ba đầu tiên.
STOOGE-SORT(A, 1 + k, j)     ▷ Hai phần ba cuối.
STOOGE-SORT(A, i, j - k)     ▷ Lại hai phần ba đầu.
```

a. Biện luận STOOGE-SORT(A, 1, *length*[A]) sắp xếp đúng đắn mảng đầu vào $A[1..n]$, ở đó $n = length[A]$.

b. Nêu một phép truy toán cho thời gian thực hiện trường hợp xấu nhất của STOOGE-SORT và một cận sát tiệm cận (hệ ký hiệu-Θ) trên thời gian thực hiện trường hợp xấu nhất.

c. So sánh thời gian thực hiện trường hợp xấu nhất của STOOGE-

SORT với của thuật toán sắp xếp chèn, sắp xếp trộn, sắp xếp đống, và sắp xếp nhanh. Các giáo sư có xứng danh không?

8-4 Chiều sâu ngăn xếp cho sắp xếp nhanh

Thuật toán QUICKSORT trong Đoạn 8.1 chứa hai lệnh gọi đệ quy đến chính nó. Sau lệnh gọi đến PARTITION, mảng con trái được sắp xếp theo đệ quy rồi mảng con phải được sắp xếp theo đệ quy. Lệnh gọi đệ quy thứ hai trong QUICKSORT thực tế không cần thiết; để tránh nó, ta có thể dùng một cấu trúc điều khiển lặp lại. Kỹ thuật này, có tên ***đệ quy đuôi*** [tail recursion], được tự động cung cấp bởi các bộ biên dịch tốt. Xem xét phiên bản dưới đây của thuật toán sắp xếp nhanh, mô phỏng đệ quy đuôi.

QUICKSORT' (A, p, r)

while $p < r$

do ▷ Phân hoạch và sắp xếp mảng con trái

$q \leftarrow$ PARTITION(A, p, r)

QUICKSORT'(A, p, q)

$p \leftarrow q + 1$

a. Chứng tỏ QUICKSORT'$(A, 1, length[A])$ sắp xếp đúng đắn mảng A.

Các trình biên dịch thường thi hành các thủ tục đệ quy bằng cách dùng một ***ngăn xếp*** [stack] chứa thông tin thích đáng, kể cả các giá trị tham số, cho từng lệnh gọi đệ quy. Thông tin của lệnh gọi mới nhất nằm trên đầu ngăn xếp, và thông tin của lệnh gọi ban đầu nằm ở cuối. Khi một thủ tục được triệu gọi, thông tin của nó ***được đẩy*** [pushed] lên ngăn xếp; khi nó kết thúc, thông tin được ***kéo ra*** [popped]. Bởi ta mặc nhận các tham số mảng thực tế được biểu thị bởi các biến trỏ [pointers], thông tin của mỗi lệnh gọi thủ tục trên ngăn xếp sẽ yêu cầu $O(1)$ không gian ngăn xếp. ***Chiều sâu ngăn xếp*** là lượng không gian ngăn xếp tối đa tại một thời điểm bất kỳ trong khi tính toán.

b. Mô tả bối cảnh ở đó chiều sâu ngăn xếp của QUICKSORT' là $\Theta(n)$ trên một mảng đầu vào n-thành phần.

c. Sửa đổi mã của QUICKSORT' sao cho chiều sâu ngăn xếp trường hợp xấu nhất là $\Theta(\lg n)$.

8-5 Phân hoạch trung tuyến-của-3

Một cách để cải thiện thủ tục RANDOMIZED-QUICKSORT đó là phân hoạch quanh một thành phần x được chọn cẩn thận hơn bằng cách lấy một thành phần ngẫu nhiên từ mảng con. Một cách tiếp cận chung

đó là phương pháp ***trung tuyến-của-3*** [median-of-3]: chọn x làm trung tuyến (thành phần giữa) của một tập hợp 3 thành phần được lựa ngẫu nhiên từ mảng con. Với bài toán này, ta mặc nhận các thành phần trong mảng đầu vào $A[1..n]$ là riêng biệt và $n \geq 3$. Ta thể hiện mảng kết xuất đã sắp xếp bằng $A'[1..n]$. Dùng phương pháp trung tuyến-của-3 để chọn thành phần trục x, định nghĩa $p_i = \Pr\{x = A'[i]\}$.

a. Nêu một công thức chính xác với *pi* làm một hàm của n và i với $i = 2, 3,..., n - 1$. (Lưu ý, $p_1 = p_n = 0$.)

b. Ta đã tăng khả năng khả !ĩ đến mức nào để chọn $x = A'[\lfloor(n + 1)/2\rfloor]$, trung tuyến của $A[1..n]$, được so sánh với tiến trình thực thi bình thường? Giả sử $n \to \infty$, và nêu tỷ số giới hạn của các xác suất này.

c. Nếu ta định nghĩa một đợt tách "tốt" để chỉ việc chọn $x = A'[i]$, ở đó $n/3 \leq i \leq 2n/3$, ta đã tăng khả năng khả dĩ đến mức nào để có một đợt tách tốt so sánh với tiến trình thực thi bình thường? (*Mách nước:* Lấy xấp xỉ tổng bằng một tích phân.)

d. Chứng tỏ phương pháp trung tuyến-của-3 chỉ ảnh hưởng đến thừa số bất biến trong $\Omega(n \lg n)$ thời gian thực hiện của sắp xếp nhanh.

Ghi chú Chương

Thủ tục sắp xếp nhanh đã được phát minh bởi Hoare [98]. Sedgewick [174] cung cấp một nguồn tham khảo tốt về chi tiết thực thi và tầm ý nghĩa của chúng. Ưu điểm của các thuật toán ngẫu nhiên hóa đã được Rabin [165] nêu rõ.

9 Sắp Xếp trong Thời Gian Tuyến Tính

Đến đây ta đã tìm hiểu vài thuật toán có thể sắp xếp n con số trong $O(n \lg n)$ thời gian. Sắp xếp trộn và sắp xếp đống hoàn thành cận trên này trong trường hợp xấu nhất; sắp xếp nhanh hoàn thành nó theo trường hợp trung bình. Hơn nữa, với từng thuật toán đó, ta có thể tạo một dãy n con số nhập khiến thuật toán chạy trong $\Omega(n \lg n)$ thời gian.

Các thuật toán này chia sẻ một tính chất thú vị: *thứ tự sắp xếp chúng xác định chỉ dựa trên các phép so sánh giữa các thành phần nhập.* Ta gọi các thuật toán sắp xếp như vậy là ***các kiểu sắp xếp so sánh***. Tất cả các thuật toán sắp xếp đã được giới thiệu cho đến giờ đều là các kiểu sắp xếp so sánh.

Trong Đoạn 9.1, ta sẽ chứng minh bất kỳ kiểu sắp xếp so sánh nào đều phải thực hiện $\Omega(n \lg n)$ phép so sánh trong trường hợp xấu nhất để sắp xếp một dãy n thành phần. Như vậy, sắp xếp trộn và sắp xếp đống là tối ưu theo tiệm cận, và không có kiểu sắp xếp so sánh nào tồn tại lại nhanh hơn theo hơn một thừa số bất biến.

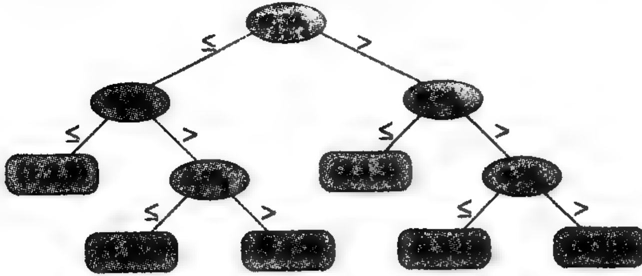
Các Đoạn 9.2, 9.3, và 9.4 xem xét ba thuật toán sắp xếp—sắp xếp đếm, sắp xếp cơ số, và sắp xếp thùng—chạy trong thời gian tuyến tính. Chẳng cần phải nói, các thuật toán này dùng các phép toán khác với các phép so sánh để xác định thứ tự sắp xếp. Bởi vậy, cận dưới $\Omega(n \lg n)$ không áp dụng cho chúng.

9.1 Các cận dưới để sắp xếp

Trong kiểu sắp xếp so sánh, ta chỉ dùng các phép so sánh giữa các thành phần để có được thông tin thứ tự về một dãy đầu vào $\langle a_1, a_2, \ldots, a_n \rangle$. Nghĩa là, căn cứ vào hai thành phần a_i và a_j, ta thực hiện một trong số các trắc nghiệm $a_i < a_j$, $a_i \le a_j$, $a_i = a_j$, $a_i \ge a_j$, hoặc $a_i > a_j$ để xác định thứ tự tương đối của chúng. Ta không thể kiểm tra các giá trị của các thành phần hoặc lấy thông tin thứ tự về chúng theo bất kỳ cách nào khác.

Trong đoạn này, ta mặc nhận mà không để mất tính tổng quát rằng

tất cả các thành phần nhập đều là riêng biệt. Căn cứ vào giả thiết này, các phép so sánh theo dạng $a_i = a_j$ đều vô dụng, do đó ta có thể mặc nhận không thực hiện phép so sánh nào theo dạng này. Cũng lưu ý, tất cả các phép so sánh $a_i \leq a_j$, $a_i \geq a_j$, $a_i > a_j$ và $a_i < a_j$ đều tương đương ở chỗ chúng cho ra thông tin giống nhau về thứ tự tương đối của a_i và a_j. Do đó ta mặc nhận tất cả các phép so sánh đều có dạng $a_i < a_j$.



Hình 9.1 Cây quyết định của sắp xếp chèn hoạt động trên ba thành phần. Có 3! = 6 phép hoán vị khả dĩ của các thành phần nhập, do đó cây quyết định phải có ít nhất 6 lá.

Mô hình cây-quyết định

Theo trừu tượng, các sắp xếp so sánh có thể được xem dưới dạng các ***cây quyết định*** [decision trees]. Một cây quyết định biểu thị cho các phép so sánh được một thuật toán sắp xếp thực hiện khi nó hoạt động trên một đầu vào có một kích cỡ nhất định. Điều khiển, dời chuyển dữ liệu, và tất cả các khía cạnh khác của thuật toán sẽ được bỏ qua. Hình 9.1 có nêu cây quyết định tương ứng với thuật toán sắp xếp chèn trong Đoạn 1.1 hoạt động trên một dãy đầu vào gồm ba thành phần.

Trong một cây quyết định, mỗi nút trong được chú thích bằng $a_i : a_j$ với một i và j nào đó trong miền giá trị $1 \leq i, j \leq n$, ở đó n là số lượng thành phần trong dãy đầu vào. Mỗi lá được chú thích bằng một phép hoán vị $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$. (Xem Đoạn 6.1 để biết về các phép hoán vị.) Việc thi hành của thuật toán sắp xếp tương ứng với việc rà theo một lộ trình từ gốc của cây quyết định đến một lá. Tại mỗi nút trong, ta thực hiện một phép so sánh $a_i \leq a_j$. Sau đó, cây con trái chỉ rõ các phép so sánh tiếp theo với $a_i \leq a_j$, và cây con phải chỉ rõ các phép so sánh tiếp theo với $a_i > a_j$. Khi ta đến một lá, thuật toán sắp xếp đã thiết lập kiểu sắp xếp thứ tự $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$. Mỗi trong số $n!$ phép hoán vị trên n thành phần phải xuất hiện như một trong số các lá của cây quyết định để thuật toán sắp xếp sắp xếp đúng đắn.

Một cận dưới cho trường hợp xấu nhất

Chiều dài của lộ trình dài nhất từ gốc của một cây quyết định đến

một lá bất kỳ của nó biểu thị cho số lượng trường hợp xấu nhất của các phép so sánh mà thuật toán sắp xếp thực hiện. Bởi vậy, số lượng trường hợp xấu nhất của các phép so sánh đối với một kiểu sắp xếp so sánh tương ứng với chiều cao của cây quyết định của nó. Do đó, một cận dưới trên các chiều cao của các cây quyết định là một cận dưới trên thời gian thực hiện của bất kỳ thuật toán sắp xếp so sánh nào. Định lý dưới đây thiết lập một cận dưới như vậy.

Định lý 9.1

Bất kỳ cây quyết định nào sắp xếp n thành phần đều có chiều cao $\Omega(n\ lg\ n)$.

Chứng minh

Xem xét một cây quyết định có chiều cao h sắp xếp n thành phần. Bởi có $n!$ phép hoán vị của n thành phần, nên mỗi phép hoán vị biểu thị cho một thứ tự sắp xếp riêng biệt, cây phải có ít nhất $n!$ lá. Bởi một cây nhị phân có chiều cao h không có hơn 2^h lá, nên ta có

$$n! \le 2^h,$$

mà, bằng cách lấy lôga, nó hàm ý

$$h \ge \lg(n!),$$

bởi hàm lg tăng đơn điệu. Từ phép xấp xỉ Stirling (2.11), ta có

$$n! > \left(\frac{n}{e}\right)^n,$$

ở đó $e = 2.71828...$là cơ số của lôga tự nhiên; như vậy

$$\begin{aligned} h &\ge \lg\left(\frac{n}{e}\right)^n \\ &= n \lg n - n \lg e \\ &= \Omega(n \lg n). \end{aligned}$$

Hệ luận 9.2

Sắp xếp đống và sắp xếp trộn là các kiểu sắp xếp so sánh tối ưu theo tiệm cận.

Chứng minh Các cận trên $O(n \lg n)$ trên các thời gian thực hiện của thuật toán sắp xếp đống và sắp xếp trộn khớp với cận dưới trường hợp xấu nhất $\Omega(n \lg n)$ từ Định lý 9.1.

Bài tập

9.1-1

Nêu chiều sâu khả dĩ nhỏ nhất của một lá trong một cây quyết định

của một thuật toán sắp xếp?

9.1-2

Lấy các cận sát tiệm cận trên lg(n!) mà không dùng phép xấp xỉ Stirling. Thay vì thế, đánh giá phép lấy tổng $\sum_{k=1}^{n} \lg k$ bằng các kỹ thuật trong Đoạn 3.2.

9.1-3

Chứng tỏ không có kiểu sắp xếp so sánh có thời gian thực hiện là tuyến tính với ít nhất phân nửa $n!$ đầu vào có chiều dài n. Còn phân số của $1/n$ của các đầu vào có chiều dài n thì sao? Một phân số $1/2^n$ thì sao?

9.1-4

Giáo sư Solomon lập luận rằng cận dưới $\Omega(n \lg n)$ để sắp xếp n con số không áp dụng cho môi trường máy tính của ông, ở đó luồng điều khiển của một chương trình có thể tách theo ba cách sau một phép so sánh đơn lẻ $a_i : a_j$, theo $a_i < a_j$, $a_i = a_j$, hoặc $a_i > a_j$. Chứng tỏ giáo sư đã sai bằng cách chứng minh rằng số lượng phép so sánh ba-cách cần thiết để sắp xếp n thành phần vẫn là $\Omega(n \lg n)$.

9.1-5

Chứng minh cần có $2n - 1$ phép so sánh trong trường hợp xấu nhất để trộn hai danh sách đã sắp xếp chứa n thành phần cho mỗi danh sách.

9.1-6

Cho một dãy n thành phần để sắp xếp. Dãy đầu vào bao gồm n/k dãy con, mỗi dãy chứa k thành phần. Tất cả các thành phần trong một dãy con đã cho nhỏ hơn các thành phần trong dãy con tiếp theo và lớn hơn các thành phần trong dãy con trước đó. Như vậy, để sắp xếp nguyên cả dãy có chiều dài n, ta chỉ cần sắp xếp k thành phần trong từng trong số n/k dãy con. Chứng tỏ cần có một cận dưới $\Omega(n \lg k)$ trên số lượng phép so sánh để giải biến thể này của bài toán sắp xếp. (*Mách nước:* Không cứng nhắc đơn thuần tổ hợp các cận dưới với các dãy con riêng lẻ.)

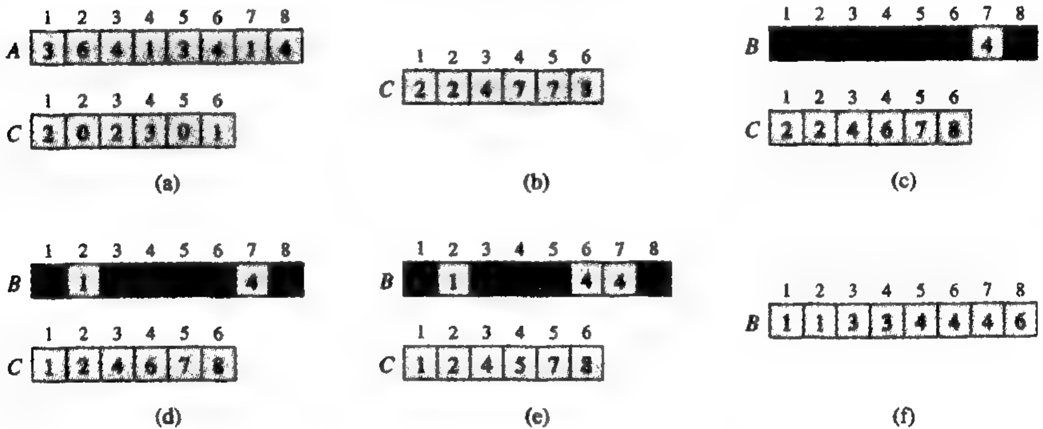
9.2 Sắp xếp đếm

Sắp xếp đếm [counting sort] mặc nhận mỗi trong số n thành phần nhập là một số nguyên trong miền giá trị 1 đến k, với một số nguyên k nào đó. Khi $k = O(n)$, tiến trình sắp xếp chạy trong $O(n)$ thời gian.

Ý tưởng cơ bản của sắp xếp đếm đó là xác định số lượng thành phần

nhỏ hơn *x*, với mỗi thành phần nhập *x*. Có thể dùng thông tin này để trực tiếp đặt thành phần *x* vào vị trí của nó trong mảng kết xuất. Ví dụ, nếu có 17 thành phần nhỏ hơn *x*, thì *x* thuộc về tại vị trí kết xuất 18. Sơ đồ này phải được sửa đổi chút đỉnh để điều quản tình huống ở đó có vài thành phần có cùng giá trị, bởi ta không muốn đặt tất cả trong cùng một vị trí.

Trong mã của thủ tục sắp xếp đếm, ta mặc nhận đầu vào là một mảng *A*[1..*n*], và như vậy *length*[*A*] = *n*. Ta yêu cầu hai mảng khác: mảng *B*[1..*n*] lưu giữ kết xuất đã sắp xếp, và mảng *C*[1..*k*] cung cấp kho lưu trữ làm việc tạm.



Hình 9.2 Phép toán của COUNTING-SORT trên một mảng đầu vào *A*[1..8], ở đó mỗi thành phần của *A* là một số nguyên dương không lớn hơn *k* = 6. **(a)** Mảng *A* và mảng phụ trợ *C* sau dòng 4. **(b)** Mảng *C* sau dòng 7. **(c)-(e)** Mảng kết xuất *B* và mảng phụ trợ *C* sau một, hai, và ba lần lặp của vòng lặp trong các dòng 9-11, theo thứ tự nêu trên. Chỉ các thành phần tô bóng sáng của mảng *B* được điền. **(f)** Mảng kết xuất đã sắp xếp chung cuộc *B*.

```
COUNTING-SORT(A, B, k)
1  for i ← 1 to k
2       do C[i] ← 0
3  for j ← 1 to length[A]
4       do C[A[j]] ← C[A[j]] + 1
5  ▷ C[i] giờ đây chứa số lượng thành phần bằng với i.
6  for ← 2 to k
7       do C[i] ← C[i] + C[i - 1]
8  ▷ C[i] giờ đây chứa số lượng thành phần nhỏ hơn hoặc bằng với i.
```

```
9   for j ← length[A] downto 1
10      do B[C[A[j]]] ← A[j]
11          C[A[j]] ← C[A[j]] - 1
```

Hình 9.2 minh họa thủ tục sắp xếp đếm. Sau khi khởi tạo trong các dòng 1-2, ta kiểm tra từng thành phần nhập trong các dòng 3-4. Nếu giá trị của một thành phần nhập là i, ta tăng $C[i]$. Như vậy, sau các dòng 3-4, $C[i]$ lưu giữ số lượng thành phần nhập bằng với i với mỗi số nguyên $i = 1, 2, ..., k$. Trong các dòng 6-7, ta xác định với mỗi $i = 1, 2, ..., k$, số lượng thành phần nhập nhỏ hơn hoặc bằng với i; điều này được thực hiện bằng cách giữ một tổng tới [running sum] của mảng C.

Cuối cùng, trong các dòng 9-11, ta đặt từng thành phần $A[j]$ vào đúng vị trí sắp xếp của nó trong mảng kết xuất B. Nếu tất cả n thành phần đều riêng biệt, thì khi ta nhập dòng 9 lần đầu tiên, với mỗi $A[j]$, giá trị $C[A[j]]$ là vị trí chung cuộc đúng đắn của $A[j]$ trong mảng kết xuất, bởi có $C[A[j]]$ thành phần nhỏ hơn hoặc bằng với $A[j]$. Bởi các thành phần có thể không riêng biệt, nên ta giảm số $C[A[j]]$ mỗi lần ta đặt một giá trị $A[j]$ vào mảng B; điều này khiến thành phần nhập kế tiếp có một giá trị bằng với $A[j]$, nếu như tồn tại, để đi đến vị trí ngay trước $A[j]$ trong mảng kết xuất.

Sắp xếp đếm yêu cầu thời gian bao lâu? Vòng lặp **for** của các dòng 1-2 mất một thời gian $O(k)$, vòng lặp **for** của các dòng 3-4 mất một thời gian $O(n)$, vòng lặp **for** của các dòng 6-7 mất một thời gian $O(k)$, và vòng lặp **for** của các dòng 9-11 mất một thời gian $O(n)$. Như vậy, tổng thời gian là $O(k + n)$. Trong thực tế, ta thường dùng kiểu sắp xếp đếm khi có $k = O(n)$, ở đó thời gian thực hiện là $O(n)$.

Sắp xếp đếm đánh bại cận dưới của $\Omega(n \lg n)$ đã chứng minh trong Đoạn 9.1 bởi vì nó không phải là một kiểu sắp xếp so sánh. Thực tế, không có các phép so sánh giữa các thành phần nhập xảy ra tại bất kỳ đâu trong mã. Thay vì thế, sắp xếp đếm sử dụng các giá trị thực tế của các thành phần để lập chỉ mục vào một mảng. Ta không áp dụng cận dưới $\Omega(n \lg n)$ để sắp xếp khi rời khỏi mô hình sắp xếp so sánh.

Một tính chất quan trọng của sắp xếp đếm đó là tính ***ổn định*** [stable]: các con số có cùng giá trị xuất hiện trong mảng kết xuất theo cùng thứ tự như trong mảng đầu vào. Nghĩa là, các mối ràng buộc giữa hai con số bị phá vỡ bởi quy tắc cho rằng bất kỳ con số nào xuất hiện đầu tiên trong mảng đầu vào cũng sẽ xuất hiện đầu tiên trong mảng kết xuất. Tất nhiên, tính chất ổn định chỉ quan trọng khi dữ liệu vệ tinh được mang đi cùng với thành phần đang được sắp xếp. Đoạn sau sẽ giải thích tại sao tính ổn định lại quan trọng.

Bài tập

9.2-1

Dùng Hình 9.2 làm mô hình, minh họa phép toán của COUNTING-SORT trên mảng A = <7,1, 3, 1, 2, 4, 5, 7, 2, 4, 3>.

9.2-2

Chứng minh COUNTING-SORT là ổn định.

9.2-3

Giả sử vòng lặp **for** trong dòng 9 của thủ tục COUNTING-SORT được viết lại là:

9 **for** $j \leftarrow 1$ **to** *length*[A]

Chứng tỏ thuật toán vẫn làm việc đúng đắn. Thuật toán đã sửa đổi có ổn định không?

9.2-4

Giả sử kết xuất của thuật toán sắp xếp là một luồng dữ liệu như một màn hình đồ họa. Sửa đổi COUNTING-SORT để tạo kết xuất theo thứ tự sắp xếp mà không dùng thêm kho lưu trữ nào đáng kể ngoài kho trong A và C. (*Mách nước:* Nối kết các thành phần của A có cùng khóa vào các danh sách. Đâu là nơi "tự do" để lưu giữ các biến trỏ cho danh sách nối kết?)

9.2-5

Cho n số nguyên trong miền giá trị 1 đến k, mô tả một thuật toán xử lý trước đầu vào của nó rồi đáp lại các phép truy vấn về bao nhiêu trong số n số nguyên rơi vào một miền giá trị $[a..b]$ trong thời gian $O(1)$. Thuật toán của bạn sẽ dùng thời gian tiền xử lý $O(n + k)$.

9.3 Sắp xếp cơ số

Sắp xếp cơ số [radix sort] là thuật toán được các máy sắp xếp lá bài sử dụng mà giờ đây bạn chỉ còn có thể tìm thấy chúng trong các bảo tàng máy tính. Các lá bài được tổ chức thành 80 cột, và trong mỗi cột có thể đục một lỗ tại một trong 12 vị trí. Máy xếp có thể được "lập trình" theo cơ học để xem xét một cột nhất định của từng lá bài trong một cỗ bài và phân phối lá bài vào một trong 12 giỏ tùy thuộc vào vị trí đã được đục lỗ. Sau đó, một thừa tác viên có thể thu thập các lá bài theo từng giỏ, sao cho các lá bài có vị trí đầu tiên được đục lỗ sẽ nằm trên đầu các lá bài có vị trí thứ hai được đục lỗ, và vân vân.

Với các chữ số thập phân, chỉ có 10 vị trí được dùng trong mỗi cột. (Hai vị trí kia được dùng để mã hóa các ký tự phi số.) Vậy một con số có *d*-chữ số sẽ choán một trường gồm *d* cột. Bởi máy xếp lá bài chỉ có thể xem xét mỗi lần một cột, nên bài toán sắp xếp *n* lá bài trên một con số *d*-chữ số yêu cầu một thuật toán sắp xếp.

Theo trực giác, ta phải sắp xếp các con số dựa trên chữ số *quan trọng nhất* của chúng, sắp xếp từng giỏ kết quả theo đệ quy, rồi tổ hợp các cỗ bài theo thứ tự. Đáng tiếc, do các lá bài trong 9 trong số 10 giỏ phải được chừa riêng để sắp xếp mỗi giỏ, nên thủ tục này phát sinh nhiều xấp lá bài trung gian phải được theo dõi. (Xem Bài tập 9.3-5.)

Sắp xếp cơ số giải quyết bài toán sắp xếp lá bài phản trực giác bằng cách sắp xếp dựa trên chữ số *ít quan trọng nhất* trước. Sau đó, các lá bài được tổ hợp thành chỉ một cỗ bài, với các lá bài trong giỏ 0 đứng trước các lá bài trong giỏ 1 đứng trước các lá bài trong giỏ 2, vân vân. Sau đó nguyên cả cỗ bài được sắp xếp lại dựa trên chữ số thứ hai ít quan trọng nhất và được tái tổ hợp giống như vậy. Tiến trình tiếp tục cho đến khi các lá bài được sắp xếp trên tất cả *d* chữ số. Đặc biệt, vào lúc đó các lá bài được sắp xếp đầy đủ trên con số có *d*-chữ số. Như vậy, chỉ cần *d* lần đi qua cỗ bài để sắp xếp. Hình 9.3 có nêu cách hoạt động của thủ tục sắp xếp cơ số trên một "cỗ bài" gồm bảy con số có 3-chữ số.

Điều thiết yếu là các kiểu sắp xếp chữ số trong thuật toán này là ổn định. Tiến trình sắp xếp được máy xếp lá bài thực hiện là ổn định, nhưng thừa tác viên phải thận trọng không được làm thay đổi thứ tự của các lá bài khi chúng xuất ra khỏi một giỏ, cho dù tất cả các lá bài trong một giỏ có cùng chữ số trong cột đã chọn.

Trong một máy tính điển hình, là một máy truy cập ngẫu nhiên-tuần tự, kỹ thuật sắp xếp cơ số đôi lúc được dùng để sắp xếp các khoản tin được lập khóa bởi nhiều trường. Ví dụ, có thể ta muốn sắp xếp ngày theo ba khóa: năm, tháng, và ngày. Có thể chạy một thuật toán sắp xếp với một hàm so sánh để, căn cứ vào hai ngày, so sánh các năm, và nếu có một mối ràng buộc [tie], so sánh các tháng, và nếu xảy ra một mối ràng buộc khác, so sánh các ngày. Một cách khác, ta có thể sắp xếp thông tin ba lần với một tiến trình sắp xếp ổn định: trước tiên là ngày, kế tiếp là tháng, và cuối cùng là năm.

329		720		720		329
457		355		329		355
657		436		436		436
839	⇒	457	⇒	839	⇒	457
436		657		355		657

720	329	457	720
355	839	657	839
	↑	↑	↑

Hình 9.3 Phép toán của thuật toán sắp xếp cơ số trên một danh sách bảy con số 3-chữ số. Cột đầu tiên là đầu vào. Các cột còn lại nêu danh sách sau các lần sắp xếp tiếp theo dựa trên các vị trí quan trọng tăng dần. Các mũi tên dọc nêu rõ vị trí chữ số được sắp xếp để tạo từng danh sách từ danh sách trước đó.

Mã của sắp xếp cơ số khá đơn giản. Thủ tục dưới đây mặc nhận mỗi thành phần trong mảng n-thành phần A có d chữ số, ở đó chữ số 1 là chữ số cấp thấp nhất và chữ số d chữ số cấp cao nhất.

RADIX-SORT(A, d)

1 **for** $i \leftarrow 1$ **to** d

2 **do** dùng một tiến trình sắp xếp ổn định để sắp xếp mảng A trên chữ số i

Tính đúng đắn của sắp xếp cơ số tùy thuộc vào phép quy nạp trên cột đang được sắp xếp (xem Bài tập 9.3-3). Việc phân tích thời gian thực hiện tùy thuộc vào kiểu sắp xếp ổn định được dùng làm thuật toán sắp xếp trung gian. Khi mỗi chữ số nằm trong miền giá trị 1 đến k, và k không quá lớn, thuật toán sắp xếp đếm ắt là chọn lựa hiển nhiên. Vậy mỗi lượt qua trên n con số có d-chữ số mất một thời gian $\Theta(n+k)$. Có d lượt qua, do đó tổng thời gian cho sắp xếp cơ số là $\Theta(dn + kd)$. Khi d bất biến và $k = O(n)$, thuật toán sắp xếp cơ số chạy trong thời gian tuyến tính.

Vài nhà khoa học máy tính thích xem số lượng bit trong một từ [word] máy tính là $\Theta(\lg n)$. Để cụ thể, ta nói rằng $d \lg n$ là số lượng bit, ở đó d là một hằng dương. Vậy, nếu mỗi con số sẽ được sắp xếp vừa trong một từ máy tính, ta có thể xem nó như một con số có d-chữ số trong hệ ký hiệu radix-n. Để có ví dụ cụ thể, hãy xét tiến trình sắp xếp 1 triệu con số 64-bit. Nhờ xử lý các con số này dưới dạng bốn-chữ số, các con số radix-2^{16}, ta có thể sắp xếp chúng trong chỉ bốn lượt qua bằng kỹ thuật sắp xếp cơ số. Điều này so sánh thuận lợi với một sắp xếp so sánh $\Theta(n \lg n)$ điển hình, yêu cầu xấp xỉ $\lg n = 20$ phép toán trên mỗi con số sẽ được sắp xếp. Đáng tiếc, phiên bản sắp xếp cơ số sử dụng sắp xếp đếm làm bước sắp xếp ổn định trung gian lại không sắp xếp tại chỗ, như nhiều kiểu sắp xếp so sánh $\Theta(n \lg n)$ vẫn làm. Như vậy, khi kho lưu trữ bộ nhớ chính khan hiếm, ta nên dùng một thuật toán như sắp xếp nhanh.

Bài tập

9.3-1

Dùng Hình 9.3 làm mô hình, minh họa phép toán của RADIX-SORT trên danh sách các từ tiếng Anh sau: COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX.

9.3-2

Thuật toán sắp xếp nào dưới đây là ổn định: sắp xếp chèn, sắp xếp trộn, sắp xếp đống, và sắp xếp nhanh? Nêu một lược đồ đơn giản tạo bất kỳ thuật toán sắp xếp nào trở thành ổn định. Nêu số lượng không gian và thời gian bổ sung mà lược đồ của bạn yêu cầu?

9.3-3

Dùng phương pháp quy nạp để chứng minh kỹ thuật sắp xếp cơ số làm việc. Ở đâu phép chứng minh của bạn cần đến giả thiết cho rằng sắp xếp trung gian là ổn định?

9.3-4

Nêu cách sắp xếp n số nguyên trong miền giá trị 1 đến n^2 trong $O(n)$ thời gian.

9.3-5 *

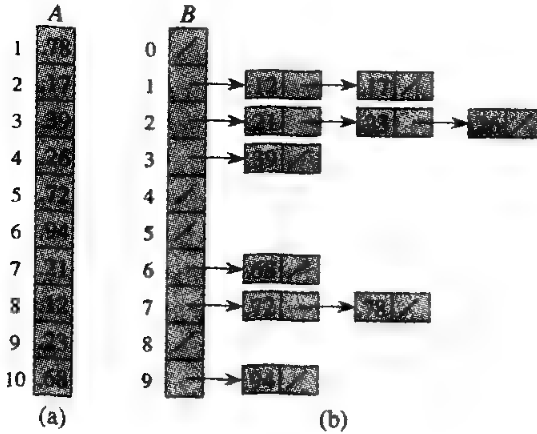
Trong thuật toán sắp xếp lá bài đầu tiên trong đoạn này, ta cần chính xác bao nhiêu lượt sắp xếp để sắp xếp các con số thập phân d-chữ số trong trường hợp xấu nhất? Một thừa tác viên cần bao nhiêu xấp lá bài để theo dõi trong trường hợp xấu nhất?

9.4 Sắp xếp thùng

Sắp xếp thùng [bucket sort] chạy trong thời gian tuyến tính theo trung bình. Giống như sắp xếp đếm, sắp xếp thùng chạy nhanh bởi nó mặc nhận một cái gì đó về đầu vào. Trong khi kỹ thuật sắp xếp đếm mặc nhận đầu vào bao gồm các số nguyên trong một miền giá trị nhỏ, thuật toán sắp xếp thùng mặc nhận đầu vào được phát sinh bởi một tiến trình ngẫu nhiên phân phối các thành phần đồng đều trên khoảng [0,1). (Xem Đoạn 6.2 để có định nghĩa về phép phân phối đồng đều.)

Ý tưởng của sắp xếp thùng đó là chia khoảng [0, 1) thành n khoảng con có kích cỡ bằng nhau, hoặc các ***thùng*** [buckets], rồi phân phối n con số nhập vào các thùng. Bởi các đầu vào được phân phối đồng đều trên [0, 1), ta không dự kiến nhiều con số sẽ rơi vào mỗi thùng. Để tạo kết xuất, ta chỉ việc sắp xếp các con số trong mỗi bô rồi đi qua các thùng

theo thứ tự, liệt kê các thành phần trong mỗi thùng.



Hình 9.4 Phép toán của BUCKET-SORT. **(a)** Mảng đầu vào $A[1..10]$. **(b)** Mảng $B[0..9]$ gồm các danh sách đã sắp xếp (các thùng) sau dòng 5 của thuật toán. Thùng i lưu giữ các giá trị trong khoảng $[i/10,(i+1)/10)$. Kết xuất đã sắp xếp bao gồm phép ghép nối theo thứ tự của các danh sách $B[0]$, $B[1]$,..., $B[9]$.

Mã của sắp xếp thùng mặc nhận đầu vào là một mảng n-thành phần A và mỗi thành phần $A[i]$ trong mảng thỏa $0 \le A[i] < 1$. Mã yêu cầu một mảng phụ trợ B[0..n - 1] gồm các danh sách nối kết (các thùng) và mặc nhận có một cơ chế để duy trì các danh sách như vậy. (Đoạn 11.2 mô tả cách thực thi các phép toán cơ bản trên on các danh sách nối kết.)

```
BUCKET-SORT(A)
1  n ← length[A]
2  for i ← 1 to n
3          do insert A[i] into list B[⌊nA[i]⌋]
4  for i ← 0 to n – 1
5          do sort list B[i] with insertion sort
6  concatenate the lists B[0], B[1],..., B[n – 1] together in order
```

Hình 9.4 có nêu phép toán sắp xếp thùng trên một mảng đầu vào gồm 10 con số.

Để xem thuật toán này làm việc, ta xét hai thành phần $A[i]$ và $A[j]$. Nếu các thành phần này rơi vào cùng thùng, chúng xuất hiện theo thứ tự tương đối đúng đắn trong dãy kết xuất bởi vì thùng của chúng được sắp xếp bằng kỹ thuật sắp xếp chèn. Tuy nhiên, giả sử chúng rơi vào các thùng khác nhau. Cho các thùng này là $B[i']$ và $B[j']$, theo thứ tự nêu trên, và giả sử mà không để mất tính tổng quát rằng $i' < j'$. Khi các danh sách của B được ghép nối trong dòng 6, các thành phần của thùng

$B[i']$ xảy ra trước các thành phần của $B[j']$, và như vậy $A[i]$ đứng trước $A[j]$ trong dãy kết xuất. Do đó, ta phải chứng tỏ $A[i] \leq A[j]$. Giả sử ngược lại, ta có

$$\begin{aligned} i' &= \lfloor nA[i] \rfloor \\ &\geq \lfloor nA[j] \rfloor \\ &= j', \end{aligned}$$

là một sự mâu thuẫn, bởi $i' < j'$. Như vậy, thuật toán sắp xếp thùng sẽ làm việc.

Để phân tích thời gian thực hiện, ta nhận thấy tất cả các dòng ngoại trừ dòng 5 bỏ ra $O(n)$ thời gian trong trường hợp xấu nhất. Tổng thời gian để xem xét tất cả các thùng trong dòng 5 là $O(n)$, và do đó phần phân tích duy nhất đáng quan tâm là thời gian thực hiện của các sắp xếp chèn trong dòng 5.

Để phân tích hao phí của các thuật toán sắp xếp chèn, cho n_i là biến ngẫu nhiên thể hiện số lượng thành phần nằm trong thùng $B[i]$. Bởi sắp xếp chèn chạy trong thời gian toàn phương (xem Đoạn 1.2), nên thời gian dự trù để sắp xếp các thành phần trong thùng $B[i]$ là $\mathrm{E}\,[O(n_i^2)] = O(\mathrm{E}\,[n_i^2])$. Như vậy, tổng thời gian dự trù để sắp xếp tất cả các thành phần trong tất cả các thùng là

$$\sum_{i=0}^{n-1} O\left(\mathrm{E}\,[n_i^2]\right) = O\left(\sum_{i=0}^{n-1} \mathrm{E}[n_i^2]\right). \tag{9.1}$$

Để đánh giá phép lấy tổng này, ta phải xác định phép phân phối của mỗi biến ngẫu nhiên n_i. Ta có n thành phần và n thùng. Xác suất mà một thành phần nhất định rơi vào thùng $B[i]$ là $1/n$, bởi mỗi thùng chịu trách nhiệm $1/n$ của khoảng $[0, 1)$. Như vậy, tình huống sẽ tương tự như ví dụ tung quả bóng ở Đoạn 6.6.2: ta có n quả bóng (các thành phần) và n giỏ (các thùng), và mỗi quả bóng được ném độc lập với xác suất $p = 1/n$ rơi vào một thùng cụ thể bất kỳ. Như vậy, xác suất $n_i = k$ sẽ theo phép phân phối nhị thức $b(k; n, p)$, có số trung bình $\mathrm{E}[n_i] = np = 1$ và phương sai $\mathrm{Var}[n_i] = np(1 - p) = 1 - 1/n$. Với bất kỳ biến ngẫu nhiên X, phương trình (6.30) sẽ cho

$$\begin{aligned} \mathrm{E}[n_i^2] &= \mathrm{Var}[n_i] + \mathrm{E}^2[n_i] \\ &= 1 - \frac{1}{n} + 1^2 \\ &= 2 - \frac{1}{n} \\ &= \Theta(1). \end{aligned}$$

Dùng cận này trong phương trình (9.1), ta kết luận rằng thời gian dự

trù của tiến trình sắp xếp chèn là $O(n)$. Như vậy, nguyên cả thuật toán sắp xếp thùng chạy trong thời gian dự trù tuyến tính.

Bài tập

9.4-1

Dùng Hình 9.4 làm mô hình, minh họa phép toán của BUCKET-SORT trên mảng A = <.79, .13, .16, .64, .39, .20, .89, .53, .71, .42>.

9.4-2

Nêu thời gian thực hiện trường hợp xấu nhất của thuật toán sắp xếp thùng? Thay đổi đơn giản nào đối với thuật toán sẽ bảo toàn thời gian thực hiện dự trù tuyến tính của nó và tạo thời gian thực hiện trường hợp xấu nhất của nó $O(n \lg n)$?

9.4-3 *

Cho n điểm trong vòng tròn đơn vị, $p_i = (x_i, y_i)$, sao cho $0 < x_i^2 + y_i^2 \leq 1$ với i - 1, 2,..., n. Giả sử các điểm được phân phối đồng đều; nghĩa là, xác suất tìm một điểm trong bất kỳ vùng nào của vòng tròn sẽ tỷ lệ với diện tích của vùng đó. Thiết kế một thuật toán thời gian dự trù $\Theta(n)$ để sắp xếp n điểm theo các khoảng cách của chúng $d_i = \sqrt{x_i^2 + y_i^2}$ từ gốc. (*Mách nước:* Thiết kế kích cỡ thùng trong BUCKET-SORT để phản ánh phép phân phối đồng đều các điểm trong vòng tròn đơn vị.)

9.4-4 *

Hàm phân phối xác suất $P(x)$ với một biến ngẫu nhiên X được định nghĩa bởi $P(x) = \Pr\{X \leq x\}$. Giả sử một danh sách n con số có một hàm phân phối xác suất liên tục P tính toán được trong thời gian $O(1)$. Nêu cách sắp xếp các con số trong thời gian dự trù tuyến tính.

Các Bài Toán

9-1 Các cận dưới trường hợp trung bình trên tiến trình sắp xếp so sánh

Trong bài toán này, ta chứng minh một cận dưới $\Omega(n \lg n)$ trên thời gian thực hiện dự trù của một tiến trình sắp xếp so sánh tất định hoặc ngẫu nhiên hóa trên n đầu vào. Ta bắt đầu bằng cách xem xét một tiến trình sắp xếp so sánh tất định A với cây quyết định T_A. Ta mặc nhận mọi phép hoán vị các đầu vào của A có khả năng như nhau.

a. Giả sử mỗi lá của T_A được gán nhãn với xác suất đạt được nó căn cứ vào một đầu vào ngẫu nhiên. Chứng minh chính xác $n!$ lá được gán nhãn $1/n!$ và phần còn lại được gán nhãn 0.

b. Cho $D(T)$ thể hiện chiều dài lộ trình ngoài của một cây T; nghĩa là, $D(T)$ là tổng các chiều sâu của tất cả các lá của T. Cho T là một cây có $k > 1$ lá, và cho RT và LT là các cây con phải và trái của T. Chứng tỏ $D(T) = D(RT) + D(LT) + k$.

c. Cho $d(m)$ là giá trị tối thiểu của $D(T)$ trên tất cả các cây T có m lá. Chứng tỏ $d(k) = \min_{1 \le i \le k} \{d(i) + d(k - i) + k\}$. (*Mách nước*: Xem xét một cây T có k lá đạt được giá trị tối thiểu. Cho i là số lượng lá trong RT và $k - i$ là số lượng là trong LT.)

d. Chứng minh rằng với một giá trị đã cho của k, hàm $i \lg i + (k - i) \lg(k - i)$ được giảm thiểu tại $i = k/2$. Kết luận rằng $d(k) = \Omega(k \lg k)$.

e. Chứng minh $D(T_A) = \Omega(n! \lg(n!))$ với T_A, và kết luận thời gian dự trù để sắp xếp n thành phần là $\Omega(n \lg n)$.

Giờ đây, ta xét một tiến trình sắp xếp so sánh *ngẫu nhiên hóa B*. Ta có thể mở rộng mô hình cây-quyết định để điều quản việc ngẫu nhiên hóa bằng cách liên kết hai kiểu nút: các nút so sánh bình thường và các nút "ngẫu nhiên hóa". Nút ngẫu nhiên hóa lập mô hình một chọn lựa ngẫu nhiên theo dạng RANDOM(1, r) được tạo bởi thuật toán B; nút có r con, mỗi con có khả năng chọn lựa bằng nhau trong khi thi hành thuật toán.

f. Chứng tỏ với một tiến trình sắp xếp so sánh ngẫu nhiên hóa B bất kỳ, ở đó tồn tại một kiểu sắp xếp so sánh tất định A mà tính trung bình không thực hiện các phép so sánh nhiều hơn B.

9-2 Sắp xếp tại chỗ trong thời gian tuyến tính

a. Giả sử ta có một mảng n khoản tin dữ liệu để sắp xếp và khóa của mỗi khoản tin có giá trị 0 hoặc 1. Nêu một thuật toán thời gian tuyến tính đơn giản để sắp xếp n khoản tin dữ liệu tại chỗ. Không dùng kho lưu trữ có kích cỡ bất biến lớn hơn ngoài kho lưu trữ mà mảng cung cấp.

b. Có thể dùng kiểu sắp xếp của bạn trong phần (a) để sắp xếp cơ số n khoản tin có các khóa b-bit trong thời gian $O(bn)$ không? Giải thích cách thực hiện hoặc tại sao không.

c. Giả sử n khoản tin có các khóa trong miền giá trị từ 1 đến k. Nêu cách sửa đổi kỹ thuật sắp xếp đếm sao cho các khoản tin có thể được sắp xếp tại chỗ trong thời gian $O(n + k)$. Có thể dùng kho lưu trữ $O(k)$ bên ngoài mảng đầu vào. (*Mách nước:* Thực hiện nó như thế nào với $k = 3$?)

Ghi chú Chương

Mô hình cây-quyết định để nghiên cứu các kiểu sắp xếp so sánh đã được Ford và Johnson [72] giới thiệu. Chuyên luận toàn diện của Knuth về sắp xếp [123] có đề cập nhiều biến thể về bài toán sắp xếp, kể cả cận dưới theo lý thuyết thông tin về tính phức tạp của sắp xếp nêu ở đây. Các cận dưới của tiến trình sắp xếp bằng các phép tổng quát hóa của mô hình cây-quyết định đã được Ben-Or [23] nghiên cứu một cách toàn diện.

Knuth cho rằng H. H. Seward đã phát minh thuật toán sắp xếp đếm vào năm 1954, và là người có ý tưởng tổ hợp sắp xếp đếm với sắp xếp cơ số. Kỹ thuật sắp xếp cơ số theo chữ số ít quan trọng nhất xuất hiện đầu tiên như một thuật toán được các thừa tác viên các máy sắp xếp lá bài cơ học sử dụng rộng rãi. Theo Knuth, tài liệu tham khảo phương pháp này được xuất bản đầu tiên là một tài liệu 1929 của L. J. Comrie mô tả thiết bị đục lỗ-lá bài. Kỹ thuật sắp xếp thùng đã được sử dụng vào khoảng năm 1956, khi ý tưởng cơ bản được E. J. Isaac và R. C. Singleton gợi ý.

10 Các Trung Bình và Thống Kê Thứ Tự

Thống kê thứ tự [order statistic] thứ i của một tập hợp n thành phần là thành phần nhỏ nhất thứ i. Ví dụ, ***cực tiểu*** [minimum] của một tập hợp các thành phần là thống kê thứ tự đầu tiên ($i = 1$), và ***cực đại*** [maximum] là thống kê thứ tự thứ n ($i = n$). Nôm na, ***trung bình*** [median] là "điểm nửa chừng" của tập hợp. Khi n là lẻ, trung bình là duy nhất, xảy ra tại $i = (n + 1) / 2$. Khi n là chẵn, ta có hai trung bình, xảy ra tại $i = n/2$ và $i = n/2 + 1$. Như vậy, bất kể tính chẵn lẻ của n, các trung bình xảy ra tại $i = \lfloor (n + 1)/2 \rfloor$ và $i = \lceil (n + 1)/2 \rceil$.

Chương này đề cập bài toán lựa thống kê thứ tự thứ i từ một tập hợp n con số riêng biệt. Để tiện dụng ta mặc nhận tập hợp chứa các con số riêng biệt, mặc dù hầu như mọi ứ1 mà ta thực hiện đều khai triển tình huống ở đó một tập hợp chứa các giá trị lặp lại. ***Bài toán lựa chọn*** có thể được đặc tả hình thức như sau:

Đầu vào: Một tập hợp A gồm n con số (riêng biệt) và một con số i, với $1 \leq i \leq n$.

Kết xuất: Thành phần $x \in A$ lớn hơn chính xác $i - 1$ thành phần khác của A.

Bài toán lựa chọn có thể giải trong thời gian $O(n \lg n)$, bởi ta có thể sắp xếp các con số bằng kỹ thuật sắp xếp đống hoặc sắp xếp trộn rồi đơn giản lập chỉ mục thành phần thứ i trong mảng kết xuất. Tuy nhiên, ta có các thuật toán nhanh hơn.

Đoạn 10.1 xem xét bài toán lựa cực tiểu và cực đại của một tập hợp các thành phần. Đáng quan tâm hơn là bài toán lựa chọn chung, được điều nghiên trong hai đoạn kế tiếp. Đoạn 10.2 phân tích một thuật toán thực tiễn đạt một cận $O(n)$ trên thời gian thực hiện trong trường hợp trung bình. Đoạn 10.3 chứa một thuật toán đáng quan tâm hơn về mặt lý thuyết hoàn thành $O(n)$ thời gian thực hiện trường hợp xấu nhất.

10.1 Cực tiểu và cực đại

Cần bao nhiêu phép so sánh để xác định cực tiểu của một tập hợp n

thành phần? Có thể dễ dàng có được một cận trên của n -1 phép so sánh: lần lượt xem xét từng thành phần của tập hợp và theo dõi thành phần nhỏ nhất đã gặp cho đến giờ. Trong thủ tục dưới đây, ta mặc nhận tập hợp thường trú trong mảng A, ở đó $length[A] = n$.

```
MINIMUM(A)
1 min ← A[1]
2 for i ← 2 to length[A]
3       do if min > A[i]
4             then min ← A[i]
5 return min
```

Tất nhiên, cũng có thể hoàn thành tiến trình tìm cực đại với n - 1 phép so sánh.

Đây có phải là điều khả dĩ tốt nhất không? Đúng, bởi ta có thể có được một cận dưới của n - 1 phép so sánh cho bài toán xác định cực tiểu. Hãy xem thuật toán xác định cực tiểu như một cuộc đấu giữa các thành phần. Mỗi phép so sánh là một trận trong cuộc đấu ở đó thành phần nhỏ hơn trong hai sẽ thắng. Nhận xét chính đó là mọi thành phần ngoại trừ "kẻ thắng" đều phải thua ít nhất một trận. Do đó, cần có n - 1 phép so sánh để xác định cực tiểu, và thuật toán *MINIMUM* là tối ưu theo số lượng phép so sánh được thực hiện.

Điểm tinh tế đáng quan tâm của phân tích đó là xác định số lần dự trù mà dòng 4 được thi hành. Bài toán 6-2 yêu cầu bạn chứng tỏ giá trị kỳ vọng này là $\Theta(\lg n)$.

Cực tiểu và cực đại đồng thời

Trong vài ứng dụng, ta phải tìm ra cả cực tiểu lẫn cực đại của một tập hợp n thành phần. Ví dụ, một chương trình đồ họa có thể cần định tỷ lệ một tập hợp (x, y) dữ liệu vừa trên một màn hình hiển thị chữ nhật hoặc một thiết bị kết xuất đồ họa khác. Để thực hiện, trước tiên chương trình phải xác định cực tiểu và cực đại của mỗi tọa độ.

Chẳng mấy khó khăn để nghĩ ra một thuật toán có thể tìm cả cực tiểu lẫn cực đại của n thành phần bằng $\Omega(n)$ số lượng phép so sánh tối ưu theo tiệm cận. Chỉ việc tìm cực tiểu và cực đại một cách độc lập, dùng n - 1 phép so sánh cho mỗi giá trị, với tổng $2n$ - 2 phép so sánh.

Thực tế, chỉ cần $3\lceil n/2 \rceil$ phép so sánh để tìm cả cực tiểu lẫn cực đại. Để thực hiện điều này, ta duy trì các thành phần cực tiểu và cực đại đã thấy cho đến giờ. Tuy nhiên, thay vì xử lý từng thành phần của đầu vào bằng cách so sánh nó với cực tiểu và cực đại hiện hành, với một mức

hao phí của hai phép so sánh trên mỗi thành phần, ta xử lý các thành phần theo các cặp. Trước tiên ta so sánh các cặp thành phần *với nhau* từ đầu vào, sau đó so sánh cặp nhỏ hơn với cực tiểu hiện hành và cặp lớn hơn với cực đại hiện hành, với một mực hao phí ba phép so sánh cho mọi hai thành phần.

Bài tập

10.1-1

Chứng tỏ có thể tìm ra thành phần nhỏ nhất thứ hai của n thành phần với $n + \lceil \lg n \rceil - 2$ phép so sánh trong trường hợp xấu nhất. (*Mách nước:* Cũng tìm thành phần nhỏ nhất.)

10.1-2 *

Chứng tỏ cần có $\lceil 3n/2 \rceil - 2$ phép so sánh trong trường hợp xấu nhất để tìm ra cả cực đại lẫn cực tiểu của n con số. (*Mách nước:* Xem xét có bao nhiêu con số có khả năng là cực đại hoặc cực tiểu, và nghiên cứu cách thức mà phép so sánh tác động lên các số đếm này.)

10.2 Lựa chọn trong thời gian tuyến tính dự trù

Bài toán lựa chọn chung có vẻ khó hơn bài toán đơn giản tìm một cực tiểu, song điều đáng ngạc nhiên là thời gian thực hiện tiệm cận của cả hai bài toán lại như nhau: $\Theta(n)$. Trong đoạn này, ta tìm hiểu thuật toán chia để trị của bài toán lựa chọn. Thuật toán RANDOMIZED-SELECT được lập mô hình dựa trên thuật toán sắp xếp nhanh trong Chương 8. Như trong sắp xếp nhanh, ý tưởng đó là phân hoạch mảng đầu vào theo đệ quy. Nhưng khác với sắp xếp nhanh, xử lý đệ quy cả hai phía của phân hoạch, RANDOMIZED-SELECT chỉ làm việc trên một phía của phân hoạch. Sự khác biệt này lộ rõ trong phân tích: trong khi kỹ thuật sắp xếp nhanh có một thời gian thực hiện dự trù là $\Theta(n \lg n)$, thời gian dự trù của RANDOMIZED-SELECT là $\Theta(n)$.

RANDOMIZED-SELECT sử dụng thủ tục RANDOMIZED-PARTITION đã giới thiệu trong Đoạn 8.3. Như vậy, giống như RANDOMIZED-QUICKSORT, nó là một thuật toán ngẫu nhiên hóa, bởi cách ứng xử của nó được xác định phần nào bởi kết xuất của một bộ phát sinh ngẫu số. Mã dưới đây của RANDOMIZED-SELECT trả về thành phần nhỏ nhất thứ i của mảng $A[p..r]$.

RANDOMIZED-SELECT (A, p, r, i)

1 **if** $p = r$

```
2        then return A[p]
3 q ← RANDOMIZED-PARTITION(A, p, r)
4 k ← q - p + 1
5 if i ≤ k
6 then return RANDOMIZED-SELECT (A, p, q, i)
7 else return RANDOMIZED-SELECT(A, q + 1, r, i - k)
```

Sau khi thi hành RANDOMIZED-PARTITION trong dòng 3 của thuật toán, mảng $A[p..r]$ được phân hoạch thành hai mảng con không trống $A[p..q]$ và $A[q + 1..r]$ sao cho mỗi thành phần của $A[p..q]$ nhỏ hơn mỗi thành phần của $A[q + 1..r]$. Dòng 4 của thuật toán tính toán số k của các thành phần trong mảng con $A[p..q]$. Giờ đây thuật toán xác định thành phần nhỏ nhất thứ i nằm trong mảng nào trong hai mảng con $A[p..q]$ và $A[q + 1..r]$. Nếu $i < k$, thì thành phần mong muốn nằm trên phía thấp của phân hoạch, và nó được lựa theo đệ quy từ mảng con trong dòng 6. Tuy nhiên, nếu $i > k$, thì thành phần mong muốn nằm trên phía cao của phân hoạch. Bởi ta đã biết k giá trị là nhỏ hơn thành phần nhỏ nhất thứ i của $A[p..r]$—tức là, các thành phần của $A[p..q]$—thành phần mong muốn là thành phần nhỏ nhất thứ $(i - k)$ của $A[q + 1..r]$, được tìm thấy theo đệ quy trong dòng 7.

Thời gian thực hiện ca xấu nhất của RANDOMIZED-SELECT là $\Theta(n^2)$, kể cả để tìm ra cực tiểu, bởi ta có thể cực kỳ không may mắn và luôn phân hoạch quanh thành phần còn lại lớn nhất. Tuy thuật toán làm việc tốt trong trường hợp trung bình, và bởi nó được ngẫu nhiên hóa, song không đầu vào cụ thể nào gợi ra cách ứng xử ca xấu nhất.

Ta có thể được một cận trên $T(n)$ trên thời gian dự trù mà RANDOMIZED-SELECT yêu cầu trên một mảng đầu vào n thành phần như sau. Trong Đoạn 8.4 ta đã nhận xét rằng thuật toán RANDOMIZED-PARTITION tạo ra một phân hoạch mà phía thấp của nó có thành phần 1 với xác suất $2/n$ và các thành phần i với xác suất $1/n$ với $i = 2, 3, ..., n - 1$. Giả sử $T(n)$ tăng đơn điệu, trong trường hợp xấu nhất RANDOMIZED-SELECT luôn không may mắn ở chỗ thành phần thứ i được xác định là trên phía lớn hơn của phân hoạch. Như vậy, ta có phép truy toán

$$
\begin{aligned}
T(n) &\le \frac{1}{n}\left(T(\max(1, n-1) + \sum_{k=1}^{n-1} T(\max(k, n-k))\right) + O(n) \\
&\le \frac{1}{n}\left(T(n-1) + 2\sum_{k=\lceil n/2 \rceil}^{n-1} T(k)\right) + O(n) \\
&= \frac{2}{n}\sum_{k=\lceil n/2 \rceil}^{n-1} T(k) + O(n).
\end{aligned}
$$

Dòng thứ hai là do dòng thứ nhất bởi $\max(1, n - 1) = n - 1$ và

$$\max(k, n - k) = \begin{cases} k & \text{nếu } k \geq \lceil n/2 \rceil, \\ n - k & \text{nếu } k < \lceil n/2 \rceil. \end{cases}$$

Nếu n là lẻ, mỗi *tohh* $T(\lceil n/2 \rceil)$, $T(\lceil n/2 \rceil + 1)$,..., $T(n - 1)$ xuất hiện hai lần trong phép lấy tổng, và nếu n là chẵn, mỗi số hạng $T(\lceil n/2 \rceil + 1)$, $T(\lceil n/2 \rceil + 2)$,..., $T(n - 1)$ xuất hiện hai lần và số hạng $T(\lceil n/2 \rceil)$ xuất hiện một lần. Trong cả hai trường hợp, phép lấy tổng của dòng thứ nhất được định cận từ bên trên bởi phép lấy tổng của dòng thứ hai. Dòng thứ ba là do dòng thứ hai bởi trong trường hợp xấu nhất $T(n - 1) = O(n^2)$, và như vậy số hạng $\frac{1}{n}T(n - 1)$ có thể được hấp thụ với số hạng $O(n)$.

Ta giải phép truy toán bằng phép thay thế. Giả sử $T(n) \leq cn$ với một hằng c nào đó thỏa các điều kiện ban đầu của phép truy toán. Dùng giả thuyết quy nạp này, ta có

$$\begin{aligned} T(n) &\leq \frac{2}{n} \sum_{k=\lceil n/2 \rceil}^{n-1} ck + O(n) \\ &\leq \frac{2c}{n} \left(\sum_{k=1}^{n-1} k - \sum_{k=\lceil n/2 \rceil}^{n-1} k \right) + O(n) \\ &= \frac{2c}{n} \left(\frac{1}{2} (n-1)n - \frac{1}{2} \left(\left\lceil \frac{n}{2} \right\rceil - 1 \right) \left\lceil \frac{n}{2} \right\rceil \right) + O(n) \\ &\leq c(n-1) - \frac{c}{n} \left(\frac{n}{2} - 1 \right) \left(\frac{n}{2} \right) + O(n) \\ &= c \left(\frac{3}{4} n - \frac{1}{2} \right) + O(n) \\ &\leq cn, \end{aligned}$$

bởi ta có thể chọn c đủ lớn sao cho $c(n/4 + 1/2)$ chi phối số hiệu $O(n)$.

Như vậy, bất kỳ thống kê thứ tự nào, và nhất là trung bình, đều có thể được xác định theo trung bình trong thời gian tuyến tính.

Bài tập

10.2-1

Viết một phiên bản lặp lại của RANDOMIZED-SELECT.

10.2-2

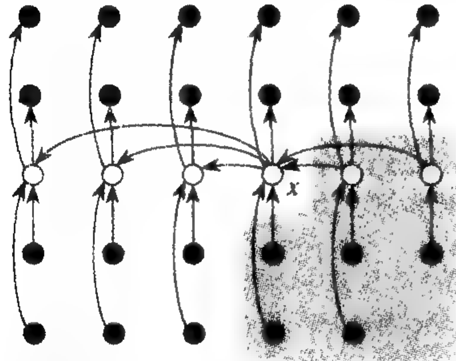
Giả sử ta dùng RANDOMIZED-SELECT để lựa thành phần cực tiểu của mảng $A = \langle 3, 2, 9, 0, 7, 5, 4, 8, 6, 1 \rangle$. Mô tả một dãy các phân hoạch dẫn đến khả năng thực hiện trong trường hợp xấu nhất của RANDOMIZED-SELECT.

10.2-3

Hãy nhớ lại, với sự hiện diện của các thành phần bằng nhau, thủ tục RANDOMIZED-PARTITION phân hoạch mảng con $A[p..r]$ thành hai mảng con không trống $A[p..g]$ và $A[q + 1..r]$ sao cho mỗi thành phần trong $A[p..q]$ nhỏ hơn *hoặc bằng với* mọi thành phần trong $A [q + 1. . r]$. Nếu có các thành phần bằng nhau, thủ tục RANDOMIZED-SELECT có làm việc đúng đắn hay không?

10.3 Lựa chọn trong thời gian tuyến tính trường hợp xấu nhất

Giờ đây ta xem xét một thuật toán lựa chọn có thời gian thực hiện là $O(n)$ trong trường hợp xấu nhất. Giống như RANDOMIZED-SELECT, thuật toán SELECT tìm thành phần mong muốn bằng cách phân hoạch đệ quy mảng đầu vào. Tuy nhiên, ý tưởng đàng sau thuật toán đó là *bảo đảm* một đợt tách tốt khi mảng được phân hoạch. SELECT sử dụng thuật toán phân hoạch tất định PARTITION của sắp xếp nhanh (xem Đoạn 8.1), được sửa đổi để đưa thành phần sẽ phân hoạch di chuyển vòng quanh dưới dạng một tham số nhập.



Hình 10.1 Phân tích thuật toán SELECT. n thành phần được biểu thị bởi các vòng tròn nhỏ, và mỗi nhóm choán một cột. Các trung bình của các nhóm được tô trắng, và trung bình-của-các trung bình x được gán nhãn. Các mũi tên được vẽ từ các thành phần lớn hơn đến nhỏ hơn, từ đó có thể thấy 3 rút từ mọi nhóm 5 thành phần về bên phải của x sẽ lớn hơn x, và 3 rút từ mọi nhóm 5 thành phần về bên trái của x sẽ nhỏ hơn x. Các thành phần lớn hơn x được nêu trên nền tô bóng.

Thuật toán SELECT xác định thành phần nhỏ nhất thứ i của một mảng đầu vào n thành phần bằng cách thi hành các bước sau.

1. Chia n thành phần của mảng đầu vào thành $\lfloor n / 5 \rfloor$ nhóm gồm 5 thành phần một và tối đa một nhóm được hình thành từ n mod 5 thành phần còn lại.

2. Tìm trung bình của mỗi trong số $\lfloor n/5 \rfloor$ nhóm bằng cách sắp xếp chèn các thành phần của mỗi nhóm (có 5 là tối đa) và lấy thành phần ở giữa của nó. (Nếu nhóm có một số chẵn các thành phần, lấy giá trị lớn hơn của hai trung bình.)

3. Dùng SELECT theo đệ quy để tìm ra trung bình x của $\lceil n/5 \rceil$ trung bình đã tìm thấy trong bước 2.

4. Phân hoạch mảng đầu vào quanh trung bình-của-các trung bình x dùng phiên bản sửa đổi của PARTITION. Cho k là số lượng các thành phần trên phía thấp của phân hoạch, sao cho $n - k$ là số lượng thành phần trên phía cao.

5. Dùng SELECT theo đệ quy để tìm ra thành phần nhỏ nhất thứ i trên phía thấp nếu $i \leq k$, hoặc thành phần nhỏ nhất thứ $(i - k)$ trên phía cao nếu $i > k$.

Để phân tích thời gian thực hiện của SELECT, trước tiên ta xác định một cận dưới trên số lượng thành phần lớn hơn thành phần phân hoạch x. Hình 10.1 tỏ ra hữu ích trong việc hình dung tiến trình theo dõi thực hiện này. Ít nhất phân nửa các trung bình tìm thấy trong bước 2 sẽ lớn hơn hoặc bằng với trung bình-của-các trung bình x. Như vậy, ít nhất phân nửa $\lceil n/5 \rceil$ nhóm đóng góp 3 thành phần lớn hơn x, ngoại trừ nhóm một có ít hơn 5 thành phần nếu 5 không chia n chính xác, và nhóm một chứa chính x. Trừ hao hai nhóm này, dẫn đến số lượng thành phần lớn hơn x chí ít sẽ là

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rceil\right) - 2 \geq \frac{3n}{10} - 6 .$$

Cũng vậy, số lượng thành phần nhỏ hơn x ít nhất là $3n/10-6$. Như vậy, trong ca xấu nhất, SELECT được gọi đệ quy trên tối đa $7n/10 + 6$ thành phần trong bước 5.

Giờ đây ta có thể phát triển một phép truy toán cho thời gian thực hiện ca xấu nhất $T(n)$ của thuật toán SELECT. Các bước 1, 2, và 4 bỏ ra $O(n)$ thời gian. (Bước 2 bao gồm $O(n)$ lwnh65 gọi của thủ tục sắp xếp chèn trên các tập hợp có kích cỡ $O(1)$.) Bước 3 mất một thời gian $T(\lceil n/5 \rceil)$, và bước 5 mất một thời gian tối đa $T(7n/10+6)$, giả định T tăng đơn điệu. Lưu ý, $7n/10 + 6 < n$ với $n > 20$ và bất kỳ đầu vào nào gồm 80 thành phần hoặc ít hơn đều yêu cầu $O(1)$ thời gian. Do đó, ta có thể có được phép truy toán

$$T(n) \leq \begin{cases} (1) & \text{nếu } n \leq 80 , \\ T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n) & \text{nếu } n > 80 . \end{cases}$$

Ta chứng tỏ thời gian thực hiện là tuyến tính bằng phép thay thế. Giả

sử $T(n) \le cn$ với một hằng c nào đó và tất cả $n \le 80$. Thay giả thiết quy nạp vào bên phía phải của phép truy toán sẽ cho ra

$$\begin{aligned} T(n) &\le c\lceil n/5\rceil + c(7n/10 + 6) + O(n) \\ &\le cn/5 + c + 7cn/10 + 6c + O(n) \\ &\le 9cn/10 + 7c + O(n) \\ &\le cn , \end{aligned}$$

bởi ta có thể chọn c đủ lớn sao cho $c(n/10 - 7)$ lớn hơn hàm mô tả bởi số hạng $O(n)$ với tất cả $n > 80$. Do đó thời gian thực hiện ca xấu nhất của SELECT là tuyến tính.

Như trong một thuật toán sắp xếp so sánh (xem Đoạn 9.1), SELECT và RANDOMIZEDSELECT chỉ xác định thông tin về thứ tự tương đối của các thành phần bằng cách so sánh các thành phần. Như vậy, cách ứng xử thời gian tuyến tính không phải là một kết quả của các giả thiết về đầu vào, như trường hợp của các thuật toán sắp xếp trong Chương 9. Tiến trình sắp xếp yêu cầu $\Omega(n \lg n)$ thời gian trong mô hình so sánh, kể cả theo trung bình (xem Bài toán 9-1), và như vậy phương pháp sắp xếp và lập chỉ mục trình bày trong phần giới thiệu chương này là không hiệu quả theo tiệm cận.

Bài tập

10.3-1

Trong thuật toán SELECT, các thành phần nhập được chia thành các nhóm 5. Thuật toán có làm việc trong thời gian tuyến tính nếu chúng được chia thành các nhóm 7 hay không? Còn các nhóm 3 thì sao?

10.3-2

Phân tích SELECT để chứng tỏ số lượng thành phần lớn hơn trung bình-của-các trung bình x và số lượng thành phần nhỏ hơn x chí ít là $\lceil n/4\rceil$ nếu $n \ge 38$.

10.3-3

Nêu cách tạo kỹ thuật sắp xếp nhanh để chạy trong $O(n \lg n)$ thời gian trong trường hợp xấu nhất.

10.3-4 *

Giả sử một thuật toán chỉ sử dụng các phép so sánh để tìm ra thành phần nhỏ nhất thứ i trong một tập hợp n thành phần. Chứng tỏ nó cũng có thể tìm ra $i - 1$ thành phần nhỏ hơn và $n - i$ thành phần lớn hơn mà không cần thực hiện thêm phép so sánh nào.

10.3-5

Căn cứ vào một chương trình con trung bình thời gian tuyến tính trường hợp xấu nhất "hộp đen", nêu mộtthuật toán thời gian tuyến tính đơn giản giải bài toán lựa chọn với một thống kê thứ tự tùy ý.

10.3-6

Các ***điểm phân vị*** [quantiles] của một tập hợp *n*-thành phần là $k-1$ thống kê thứ tự chia tập hợp đã sắp xếp thành *k* tập hợp có kích cỡ bằng nhau (đến trong 1). Nêu một thuật toán $O(n \lg k)$ thời gian để liệt kê các điểm phân vị thứ *k* của một tập hợp.

10.3-7

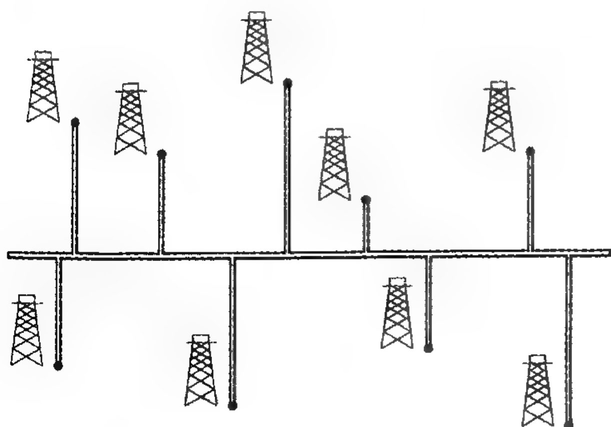
Mô tả một thuật toán $O(n)$-thời gian mà, căn cứ vào một tập hợp S n con số riêng biệt và một số nguyên dương $k \leq n$, xác định *k* con số trong *S* sát nhất với trung bình của *S*.

10.3-8

Cho $X[1..n]$ và $Y[1..n]$ là hai mảng, mỗi mảng chứa *n* con số đã được sắp xếp thứ tự. Nêu một thuật toán $O(\lg n)$-thời gian để tìm ra trung bình của tất cả $2n$ thành phần trong các mảng *X* và Y.

10.3-9

Giáo sư Olay đang tư vấn cho một công ty dầu khí, đang hoạch định một hệ thống ống dẫn dầu lớn chạy từ đông sang tây qua một bãi dầu có *n* giếng. Từ mỗi giếng dầu, một lộ trình nhánh ống dẫn phải được nối trực tiếp với đường ống dẫn chính dọc theo một lộ trình ngắn nhất (hoặc bắc hoặc nam), như đã nêu trong Hình 10.2. Căn cứ vào các tọa độ *x* và *y* của các giếng dầu, giáo sư phải chọn vị trí tối ưu của đường ống dẫn chính (đường ống giảm thiểu tổng chiều dài của các nhánh) như thế nào? Chứng tỏ vị trí tối ưu có thể được xác định trong thời gian tuyến tính.



Hình 10.2 Ta muốn xác định vị trí của đường ống dẫn dầu đông-tây giảm thiểu tổng chiều dài của các nhánh bắc-nam.

Các Bài Toán

10-1 Các số i lớn nhất trong thứ tự sắp xếp

Căn cứ vào một tập hợp n con số, ta muốn tìm ra i lớn nhất trong thứ tự sắp xếp dùng một thuật toán gốc so sánh. Tìm thuật toán thực thi mỗi phương pháp dưới đây với thời gian thực hiện tiệm cận tốt nhất trong trường hợp xấu nhất, và phân tích các thời gian thực hiện của các phương pháp theo dạng n và i.

a. Sắp xếp các con số và liệt kê i lớn nhất.

b. Xây dựng một hàng đợi ưu tiên từ các con số và gọi EXTRACT-MAX i lần.

c. Dùng một thuật toán thông kê thứ tự để tìm ra con số thứ i lớn nhất, phân hoạch, và sắp xếp i con số lớn nhất.

10-2 Trung bình gia trọng

Với n thành phần riêng biệt $x_1, x_2, \ldots, x_n$ với các trọng số [weights] dương $w_1, w_2, \ldots, w_n$ sao cho $\sum_{i=1}^{n} w_i = 1$, ***trung bình gia trọng*** [weighted median] là thành phần xk thỏa

$$\sum_{x_i<x_k} w_i \leq \frac{1}{2}$$

và

$$\sum_{x_i>x_k} w_i \leq \frac{1}{2} \, .$$

a. Biện luận trung bình của $x_1, x_2, \ldots, x_n$ là trung bình gia trọng của x_i với các trọng số $w_i = 1/n$ với $i = 1, 2, \ldots, n$.

b. Nêu cách tính trung bình gia trọng của n thành phần trong $O(n \lg n)$ thời gian trường hợp xấu nhất dùng tiến trình sắp xếp.

c. Nêu cách tính trung bình gia trọng trong $\Theta(n)$ thời gian trường hợp xấu nhất một thuật toán trung bình thời gian tuyến tính như SELECT trong Đoạn 10.3.

Bài toán vị trí bưu điện được định nghĩa như sau. Được biết n điểm $p_1, p_2, \ldots, p_n$ với các trọng số kết hợp $w_1, w_2, \ldots, w_n$. Ta muốn tìm một ra một điểm p (không nhất thiết là một trong các điểm đầu vào) giảm thiểu tổng $\sum_{i=1}^{n} w_i d(p, p_i)$, ở đó $d(a, b)$ là khoảng cách giữa các điểm a và b.

d. Chứng tỏ trung bình gia trọng là một nghiệm tốt nhất cho bài toán

vị trí bưu điện 1-chiều [1-dimensional], ở đó các điểm chẳng qua là các số thực và khoảng cách giữa các điểm a và b là $d(a, b) = |a - b|$.

e. Tìm nghiệm tốt nhất cho bài toán vị trí bưu điện 2-chiều, ở đó các điểm là (x, y) các cặp tọa độ và khoảng cách giữa các điểm $a = (x_1, y_1)$ và $b = (x_2, y_2)$ là ***khoảng cách Manhattan***: $d(a, b) = |x_1 - x_2| + |y_1 - y_2|$.

10-3 Thống kê thứ tự nhỏ

Số $T(n)$ phép so sánh trong trường hợp xấu nhất được SELECT dùng để lựa thống kê thứ tự thứ i từ n con số đã được nêu để thỏa $T(n) = \Theta(n)$, nhưng hằng mà hệ ký hiệu Θ che giấu lại khá lớn. Khi i là nhỏ tương đối với n, ta có thể thực thi một thủ tục khác sử dụng SELECT làm chương trình con nhưng thực hiện ít phép so sánh hơn trong trường hợp xấu nhất.

a. Mô tả một thuật toán sử dụng $U_i(n)$ phép so sánh để tìm ra thành phần nhỏ nhất thứ i của n thành phần, ở đó $i \le n/2$ và

$$U_i(n) = \begin{cases} T(n) & \text{nếu } n \le 2i, \\ n/2 + U_i(n/2) + T(2i) & \text{bằng không}. \end{cases}$$

(*Mách nước*. Bắt đầu với $\lfloor n/2 \rfloor$ phép so sánh từng cặp rời nhau, và đệ quy trên tập hợp chứa thành phần nhỏ hơn từ mỗi cặp.)

b. Chứng tỏ $U_i(n) = n + O(T(2i)\lg(n/i))$.

c. Chứng tỏ nếu i là một hằng, thì $U_i(n) = n + O(\lg n)$.

d. Chứng tỏ nếu $i = n/k$ for $k \ge 2$, thì $U_i(n) = n + O(T(2n/k)\lg k)$.

Ghi chú Chương

Thuật toán tìm trung bình trường hợp xấu nhất đã được phát minh bởi Blum, Floyd, Pratt, Rivest, và Tarjan [29]. Phiên bản thời gian trung bình nhanh là thuộc về Hoare [97]. Floyd và Rivest [70] đã phát triển một phiên bản thời gian trung bình cải tiến phân hoạch quanh một thành phần được lựa theo đệ quy từ một mẫu nhỏ các thành phần.

III Các Cấu Trúc Dữ Liệu

Giới thiệu

Với khoa học máy tính, các tập hợp cũng cơ bản như đối với toán học. Trong khi các tập hợp toán học không thay đổi, các tập hợp mà các thuật toán điều tác có thể tăng trưởng, co cụm, hoặc bằng không sẽ thay đổi qua thời gian. Ta gọi các tập hợp như vậy là ***động*** [dynamic]. Năm chương tiếp theo sẽ trình bày vài kỹ thuật căn bản để biểu thị các tập hợp động hữu hạn và điều tác chúng trên một máy tính.

Các thuật toán có thể yêu cầu thực hiện vài kiểu phép toán khác nhau trên các tập hợp. Ví dụ, nhiều thuật toán chỉ cần khả năng chèn các thành phần vào, xóa các thành phần ra khỏi, và trắc nghiệm tư cách phần tử [membership] trong một tập hợp. Một tập hợp động hỗ trợ các phép toán này được gọi là một ***từ điển*** [dictionary]. Các thuật toán khác yêu cầu các phép toán phức hợp hơn. Ví dụ, các hàng đợi ưu tiên, đã được giới thiệu trong Chương 7 theo ngữ cảnh cấu trúc dữ liệu đống, hỗ trợ các phép toán chèn một thành phần vào và trích thành phần nhỏ nhất từ một tập hợp. Chẳng có gì ngạc nhiên, cách tốt nhất để thực thi một tập hợp động tùy thuộc vào các phép toán phải được hỗ trợ.

Các thành phần của một tập hợp động

Trong một thực thi điển hình của một tập hợp động, mỗi thành phần được biểu thị bởi đối tượng có các trường có thể được xem xét và điều tác nếu như ta có một biến trỏ đến đối tượng. (Chương 11 sẽ mô tả cách thực thi của các đối tượng và các biến trỏ trong các môi trường lập trình không chứa chúng dưới dạng các kiểu dữ liệu cơ bản.) Có vài kiểu tập hợp động mặc nhận một trong các trường của đối tượng là một trường ***khóa*** định danh [identifying key field]. Nếu tất cả các khóa đều khác nhau, ta có thể xem tập hợp động như là một tập hợp các giá trị khóa. Đối tượng có thể chứa ***dữ liệu vệ tinh*** [satellite data], được di chuyển vòng quanh trong các trường đối tượng khác mà bằng không lại không được sử dụng bởi việc thực thi tập hợp. Cũng có thể có các trường được điều tác bởi các phép toán tập hợp; các trường này có thể chứa dữ liệu hoặc các biến trỏ đến các đối tượng khác trong tập hợp.

Có vài tập hợp động giả sử trước rằng các khóa được rút từ một tập hợp được sắp xếp hoàn toàn, như các số thực, hoặc tập hợp tất cả các từ

theo kiểu sắp xếp thứ tự bảng chữ cái bình thường. (Một tập hợp được sắp xếp hoàn toàn thỏa tính chất tam phân [trichotomy], đã được định nghĩa ở trang 31) Ví dụ, một tiến trình sắp xếp thứ tự hoàn toàn cho phép ta định nghĩa thành phần cực tiểu của tập hợp, hoặc đề cập đến thành phần kế tiếp lớn hơn một thành phần đã cho trong một tập hợp.

Các phép toán trên các tập hợp động

Các phép toán trên một tập hợp động có thể được gom nhóm thành hai phạm trù: ***các phép truy vấn*** [queries], đơn giản trả về thông tin về tập hợp, và ***các phép toán sửa đổi***, thay đổi tập hợp. Dưới đây là một danh sách các phép toán điển hình. Thông thường, các ứng dụng cụ thể chỉ yêu cầu thực thi một vài trong số các phép toán này.

SEARCH(S, k) Một phép truy vấn mà, căn cứ vào một tập hợp S và một trị khóa k, sẽ trả về một biến trỏ x đến một thành phần trong S sao cho $key[x] = k$, hoặc NIL nếu không có thành phần nào như vậy thuộc về S.

INSERT(S, x) Một phép toán sửa đổi làm tăng tập hợp S bằng thành phần được trỏ đến bởi x. Ta thường mặc nhận bất kỳ trường nào trong thành phần x được thực thi tập hợp cần đến đều đã được khởi tạo sẵn.

DELETE(S, x) Một phép toán sửa đổi mà, căn cứ vào một biến trỏ x đến một thành phần trong tập hợp S, sẽ gỡ bỏ x ra khỏi S. (Lưu ý, phép toán này sử dụng một biến trỏ đến một thành phần x, chứ không phải một trị khóa.)

MINIMUM(S) Một phép truy vấn trên một tập hợp được sắp xếp thứ tự hoàn toàn S sẽ trả về thành phần của S có khóa nhỏ nhất.

MAXIMUM(S) Một phép truy vấn trên một tập hợp được sắp xếp thứ tự hoàn toàn S sẽ trả về thành phần của S có khóa lớn nhất.

SUCCESSOR(S, x) Một phép truy vấn mà, căn cứ vào một thành phần x có khóa lấy từ một tập hợp được sắp xếp thứ tự hoàn toàn S, sẽ trả về thành phần lớn hơn kế tiếp trong S, hoặc NIL nếu x là thành phần cực đại.

PREDECESSOR(S, x) Một phép truy vấn mà, căn cứ vào một thành phần x có khóa lấy từ một tập hợp được sắp xếp thứ tự hoàn toàn S, sẽ trả về thành phần nhỏ hơn kế tiếp trong S, hoặc NIL nếu x là thành phần cực tiểu.

Các phép truy vấn SUCCESSOR và PREDECESSOR thường được khai triển ra các tập hợp có các khóa không riêng biệt [nondistinct keys]. Với một tập hợp trên n khóa, điều giả định bình thường đó là một lệnh gọi đến MINIMUM theo sau là n - 1 lệnh gọi đến SUCCESSOR điểm lại [enumerates] các thành phần trong tập hợp theo thứ tự đã sắp xếp.

Thời gian bỏ ra để thi hành một phép toán tập hợp thường được đo

theo dạng kích cỡ của tập hợp được cung cấp dưới dạng một trong các đối số của nó. Ví dụ, Chương 14 mô tả một cấu trúc dữ liệu có thể hỗ trợ bất kỳ phép toán nào nêu trên đây trên một tập hợp có kích cỡ n trong thời gian $O(\lg n)$.

Khái quát về Phần III

Các chương 11-15 mô tả vài cấu trúc dữ liệu có thể được dùng để thực thi các tập hợp động; nhiều cấu trúc sẽ được dùng về sau để kiến tạo các thuật toán hiệu quả cho các bài toán khác nhau. Một cấu trúc dữ liệu quan trọng khác—đống—đã được giới thiệu trong Chương 7.

Chương 11 mô tả căn bản cách làm việc với các cấu trúc dữ liệu đơn giản như các ngăn xếp, các hàng đợi, các danh sách nối kết, và các cây có gốc. Nó cũng nêu cách thực thi các đối tượng và các biến trỏ trong các môi trường lập trình không hỗ trợ chúng dưới dạng các căn tố [primitives]. Phần lớn bảo đảm này đều quen thuộc đối với những ai đã qua một khóa lập trình căn bản.

Chương 12 giới thiệu các bảng ánh số [hash tables], hỗ trợ các phép toán từ điển INSERT, DELETE, và SEARCH. Trong trường hợp xấu nhất, kỹ thuật ánh số yêu cầu $\Theta(n)$ thời gian để thực hiện một phép toán SEARCH, nhưng thời gian dự trù cho các phép toán ánh số là $O(1)$. Việc phân tích kỹ thuật ánh số dựa vào xác suất, nhưng phần lớn chương không yêu cầu kiến thức về chủ đề này.

Các cây tìm nhị phân, xem Chương 13, hỗ trợ tất cả các phép toán tập hợp động nêu trên đây. Trong trường hợp xấu nhất, mỗi phép toán bỏ ra $\Theta(n)$ thời gian trên một cây có n thành phần, nhưng trên một cây tìm nhị phân được xây dựng ngẫu nhiên, thời gian dự trù cho mỗi phép toán là $O(\lg n)$. Các cây tìm nhị phân được dùng làm cơ sở cho nhiều cấu trúc dữ liệu khác.

Chương 14 giới thiệu các cây đỏ đen [red-black trees], một biến thể của các cây tìm nhị phân. Khác với các cây tìm nhị phân bình thường, các cây đỏ đen được bảo đảm sẽ thực hiện tốt: các phép toán mất $O(\lg n)$ thời gian trong trường hợp xấu nhất. Một cây đỏ đen là một cây tìm cân đối; Chương 19 trình bày một kiểu cây tìm cân đối khác, có tên cây-B. Mặc dù cơ học của các cây đỏ đen có hơi phức tạp, song bạn có thể lượm lặt hầu hết các tính chất của chúng từ chương này mà không cần nghiên cứu kỹ về cơ học. Tuy vậy, tiến trình rà qua mã cũng khá bổ ích cho kiến thức.

Chương 15 đề cập cách tăng cường các cây đỏ đen để hỗ trợ các phép toán khác ngoài các phép toán căn bản nêu trên đây. Trước tiên, ta tăng cường chúng để có thể năng động duy trì thống kê thứ tự cho một tập hợp các khóa. Sau đó, tăng cường chúng theo một cách khác để duy trì các khoảng của các số thực.

11 Các cấu trúc dữ liệu cơ bản

Trong chương này, ta xem xét phần biểu thị của các tập hợp động bởi các cấu trúc dữ liệu đơn giản sử dụng các biến trỏ. Mặc dù có thể tạo dựng nhiều cấu trúc dữ liệu phức hợp bằng các biến trỏ, song ở đây ta chỉ trình bày các cấu trúc sơ đẳng: các ngăn xếp, các hàng đợi, các danh sách nối kết, và các cây có gốc. Chương này cũng mô tả một phương pháp để tổng hợp các đối tượng và các biến trỏ từ các mảng.

11.1 Các ngăn xếp và các hàng đợi

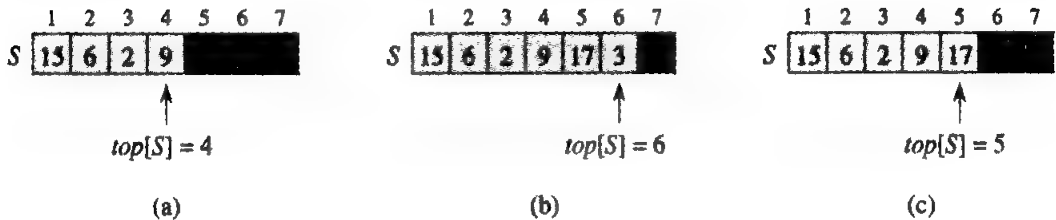
Ngăn xếp và hàng đợi là những tập hợp động ở đó ta chỉ định trước thành phần được phép toán DELETE gỡ bỏ ra khỏi tập hợp. Trong một ***ngăn xếp***, thành phần được xóa ra khỏi tập hợp chính là thành phần được chèn mới nhất: ngăn xếp thực thi nội quy ***vào sau, ra trước,*** hoặc ***LIFO*** [last-in, first out]. Cũng vậy, trong một ***hàng đợi,*** thành phần được xóa luôn là thành phần đã nằm trong tập hợp một thời gian lâu nhất: hàng đợi thực thi nội quy ***vào trước, ra trước,*** hoặc ***FIFO*** [first-in, first out]. Có vài cách hiệu quả để thực thi các ngăn xếp và các hàng đợi trên một máy tính. Đoạn này giải thích cách sử dụng một mảng đơn giản để thực thi chúng.

Các ngăn xếp

Phép toán INSERT trên một ngăn xếp thường được gọi là PUSH, và phép toán DELETE, không nhận một đối số thành phần, thường có tên là POP. Các tên này ám chỉ các ngăn xếp vật lý, chẳng hạn như các ngăn xếp đĩa được nạp bằng lò xo trong các quán ăn tự phục vụ. Thứ tự qua đó các đĩa được kéo ra khỏi ngăn xếp nghịch đảo với thứ tự mà chúng được đẩy lên ngăn xếp, bởi ta chỉ truy xuất đĩa trên cùng.

Như đã nêu trong Hình 11.1, ta có thể thực thi một ngăn xếp có tối đa n thành phần với một mảng $S[1..n]$. Mảng có một thuộc tính $top[S]$ lập chỉ mục thành phần mới được chèn. Ngăn xếp bao gồm các thành phần $S[1..top[S]]$, ở đó $S[1]$ là thành phần ở cuối ngăn xếp và $S[top[S]]$ là thành phần trên cùng.

Khi *top*[*S*] = 0, ngăn xếp không chứa thành phần nào và là ***trống***. Để kiểm tra tính rỗng của ngăn xếp, ta dùng phép toán truy vấn STACK-EMPTY. Nếu ngăn xếp trống được kéo ra, ta nói ngăn xếp ***tràn dưới*** [underflow], thường là một lỗi. Nếu *top*[*S*] vượt quá *n*, ngăn xếp ***tràn trên*** [overflow]. (Trong thực thi mã giả, ta đừng lo nghĩ về sự cố tràn trên của ngăn xếp.)



Hình 11.1 Một mảng thực thi của ngăn xếp *S*. Các thành phần ngăn xếp chỉ xuất hiện tại các vị trí tô bóng sáng. **(a)** Ngăn xếp *S* có 4 thành phần. Thành phần trên cùng là 9. **(b)** Ngăn xếp *S* sau các lệnh gọi PUSH(*S*, 17) và PUSH(*S*, 3). **(c)** Ngăn xếp *S* sau lệnh gọi POP(*S*) đã trả về thành phần 3, là thành phần được đẩy gần nhất. Mặc dù thành phần 3 vẫn xuất hiện trong mảng, song nó không còn trong ngăn xếp; trên đầu là thành phần 17.

Các phép toán ngăn xếp có thể được thực thi bằng vài dòng mã.

```
STACK-EMPTY(S)
if  top[S] = 0
    then return TRUE
    else return FALSE

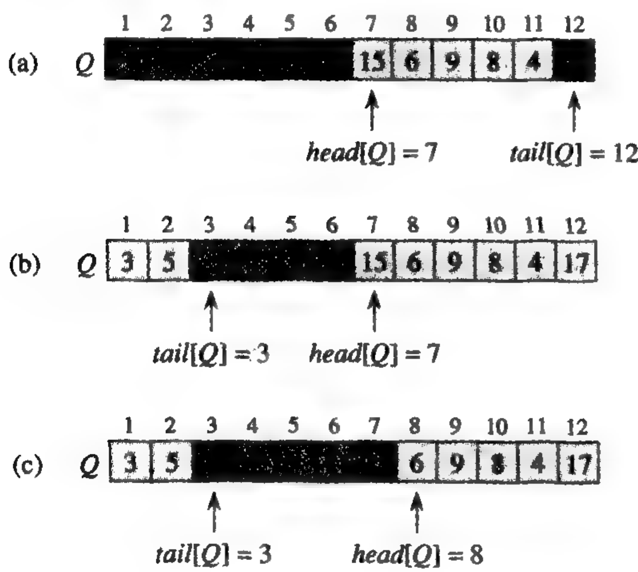
PUSH (S, x)
top[S] ← top[S] + 1
S[top[S]] ← x

POP(S)
if STACK-EMPTY(S)
    then error "underflow"
    else top[S] ← top[S] - 1
         return  S[top[S] + 1]
```

Hình 11.1 có nêu các hiệu ứng của các phép toán sửa đổi PUSH và POP. Mỗi trong số ba phép toán ngăn xếp bỏ ra $O(1)$ thời gian.

Các hàng đợi

Ta gọi phép toán INSERT trên một hàng đợi là ENQUEUE, và gọi phép toán DELETE là DEQUEUE; giống như phép toán ngăn xếp POP, DEQUEUE không nhận đối số thành phần. Tính chất FIFO của một hàng đợi sẽ khiến nó hoạt động giống như một hàng người trong một văn phòng giải quyết hồ sơ. ***Hàng đợi*** có một ***đầu*** và một ***đuôi***. Khi một thành phần được đưa vào hàng đợi [enqueued], nó chiếm vị trí tại đuôi hàng đợi, giống như một sinh viên mới đến đứng vào cuối hàng. Thành phần ra khỏi hàng đợi luôn nằm tại đầu hàng đợi, giống như sinh viên ở đầu hàng đã đợi lâu nhất. (May thay, ta không phải quan tâm về các thành phần tính toán xen vào giữa dòng.)



Hình 11.2 Một hàng đợi được thực thi bằng một mảng *Q*[1..12]. Các thành phần hàng đợi chỉ xuất hiện trong các vị trí tô bóng sáng. **(a)** Hàng đợi có 5 thành phần, trong các vị trí *Q*[7..11]. **(b)** Cấu hình của hàng đợi sau các lệnh gọi ENQUEUE(*Q*, 17), ENQUEUE(*Q*, 3), và ENQUEUE(*Q*, 5). **(c)** Cấu hình của hàng đợi sau lệnh gọi DEQUEUE(*Q*) trả về giá trị khóa 15 mà trước đó nằm tại đầu hàng đợi. Đầu mới có khóa 6.

Hình 11.2 nêu cách thực thi một hàng đợi có tối đa n - 1 thành phần dùng một mảng $Q[1..n]$. Hàng đợi có một thuộc tính *head*[*Q*] lập chỉ mục, hoặc trỏ đến, đầu của nó. Thuộc tính *tail*[*Q*] lập chỉ mục vị trí kế tiếp tại đó thành phần mới đến sẽ được chèn vào hàng đợi. Các thành phần trong hàng đợi nằm tại các vị trí *head* [*Q*], *head* [*Q*]+ 1,..., *tail*[*Q*]-1, ở đó ta "đóng khung vòng quanh" theo nghĩa vị trí 1 theo ngay sau vị trí n trong một thứ tự vòng tròn. Khi *head*[*Q*] = *tail*[*Q*], hàng đợi là

trống. Thoạt đầu, ta có *head*[*Q*] = *tail*[*Q*] = 1. Khi hàng đợi trống, nếu gắng đẩy một thành phần ra khỏi hàng đợi, hàng đợi sẽ tràn dưới. Khi *head*[*Q*] = *tail*[*Q*]+ 1, hàng đợi đầy, và nếu gắng đưa một thành phần vào hàng đợi, hàng đợi sẽ tràn trên.

Trong các thủ tục ENQUEUE và DEQUEUE ở đây, ta bỏ qua tính năng kiểm tra lỗi tràn dưới và tràn trên. (Bài tập 11.1-4 yêu cầu bạn cung cấp mã kiểm tra hai điều kiện lỗi này.)

```
ENQUEUE(Q, x)
Q[tail[Q]] ← x
if tail[Q] = length[Q]
    then tail[Q] ← 1
    else tail[Q] ← tail[Q] + 1
```

```
DEQUEUE(Q)
x ← Q[head[Q]]
if head[Q] = length[Q]
    then head [Q] ← 1
    else head[Q] ← head[Q] + 1
return x
```

Hình 11.2 nêu các hiệu ứng của các phép toán ENQUEUE và DEQUEUE. Mỗi phép toán bỏ ra $O(1)$ thời gian.

Bài tập

11.1-1

Dùng Hình 11.1 làm mô hình, minh họa kết quả của từng phép toán PUSH(*S*, 4), PUSH(*S*, 1), PUSH(*S*, 3), POP(*S*), PUSH(*S*, 8), và Pop(*S*) trên ngăn xếp *S* thoạt đầu là trống được lưu trữ trong mảng *S*[1..6].

11.1-2

Giải thích cách thực thi hai ngăn xếp trong một mảng *A*[1..*n*] sao cho không có ngăn xếp nào tràn trên trừ phi tổng các thành phần trong cả hai ngăn xếp hợp lại là *n*. Các phép toán PUSH và POP sẽ chạy trong $O(1)$ thời gian.

11.1-3

Dùng Hình 11.2 làm mô hình, minh họa kết quả của từng phép toán ENQUEUE(*Q*,4), ENQUEUE(*Q*,1), ENQUEUE(*Q*,3), DEQUEUE(*Q*), ENQUEUE(*Q*, 8), và DEQUEUE(*Q*) trên một hàng đợi *Q* thoạt đầu là trống được lưu trữ trong mảng *Q*[1..6].

11.1-4

Viết lại ENQUEUE và DEQUEUE để phát hiện sự cố tràn dưới và tràn trên của một hàng đợi.

11.1-5

Trong khi một ngăn xếp chỉ cho phép chèn và xóa các thành phần tại một đầu, và một hàng đợi cho phép chèn tại một đầu và xóa tại đầu kia, một ***đéc*** (deque = hàng đợi hai đầu) cho phép chèn và xóa tại cả hai đầu. Viết bốn thủ tục $O(1)$-thời gian để chèn các thành phần vào và xóa các thành phần ra khỏi cả hai đầu của một đéc được kiến tạo từ một mảng.

11.1-6

Nêu cách thực thi hàng đợi dùng hai ngăn xếp. Phân tích thời gian thực hiện của các phép toán hàng đợi.

11.1-7

Nêu cách thực thi một ngăn xếp dùng hai hàng đợi. Phân tích thời gian thực hiện của các phép toán ngăn xếp.

11.2 Các danh sách nối kết

Một ***danh sách nối kết*** [linked list] là một cấu trúc dữ liệu qua đó các đối tượng được dàn xếp theo thứ tự tuyến tính. Tuy vậy, khác với một mảng ở đó thứ tự tuyến tính được xác định bởi các chỉ số mảng, thứ tự trong một danh sách nối kết được xác định bởi một biến trỏ trong mỗi đối tượng. Các danh sách nối kết cung cấp một phần biểu thị đơn giản, cơ động cho các tập hợp động, hỗ trợ (tuy không nhất thiết hiệu quả) tất cả các phép toán nêu trên trang 228.

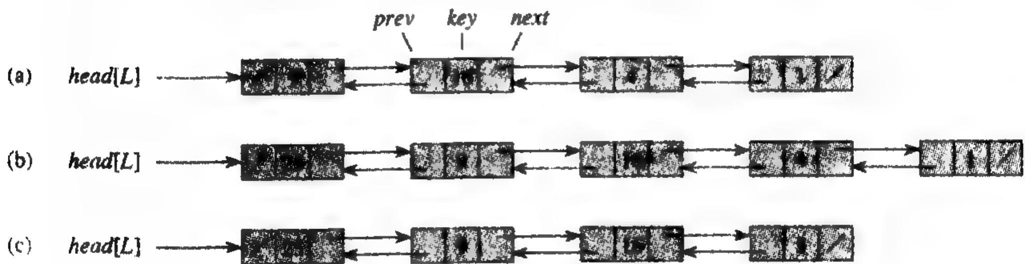
Như đã nêu trong Hình 11.3, mỗi thành phần của một ***danh sách nối kết đôi*** [doubly linked list] là một đối tượng có một trường *key* và hai trường biến trỏ khác: *next* và *prev*. Đối tượng cũng có thể chứa các dữ liệu vệ tinh khác. Căn cứ vào một thành phần x trong danh sách, $next[x]$ trỏ đến phần tử kế vị [successor] của nó trong danh sách nối kết, và $prev[x]$ trỏ đến phần tử tiền vị [predecessor] của nó. Nếu $prev[x]$ = NIL, thành phần x không có phần tử tiền vị và do đó là thành phần đầu tiên, hoặc ***đầu***, của danh sách. Nếu $next[x]$ = NIL, thành phần x không có phần tử kế vị và do đó là thành phần cuối, hoặc ***đuôi***, của danh sách. Thuộc tính *head* $[L]$ trỏ đến thành phần đầu tiên của danh sách. Nếu *head* $[L]$ = NIL, danh sách sẽ trống.

Một danh sách có thể có một trong số vài dạng. Nó có thể được nối

kết đơn hoặc nối kết đôi, có thể được sắp xếp hoặc không, và có thể xoay vòng hoặc không. Nếu danh sách được *nối kết đơn,* ta bỏ qua biến trỏ *prev* trong mỗi thành phần. Nếu danh sách được ***sắp xếp,*** thứ tự tuyến tính của danh sách tương ứng với thứ tự tuyến tính của các khóa lưu trữ trong các thành phần của danh sách; thành phần cực tiểu là đầu danh sách, và thành phần cực đại là đuôi. Nếu danh sách ***không sắp xếp,*** các thành phần có thể xuất hiện theo một thứ tự bất kỳ. Trong ***danh sách vòng*** [circular list], biến trỏ *prev* của đầu danh sách trỏ đến đuôi, và biến trỏ *next* của đuôi danh sách trỏ đến đầu. Như vậy, danh sách có thể được xem như một vòng khâu gồm các thành phần. Phần còn lại của đoạn này mặc nhận các danh sách mà ta đang làm việc đều không sắp xếp và nối kết đôi.

Tìm kiếm một danh sách nối kết

Thủ tục LIST-SEARCH(L, k) tìm thành phần đầu tiên có khóa k trong danh sách L bằng một đợt tìm kiếm tuyến tính đơn giản, trả về một biến trỏ đến thành phần này. Nếu không có đối tượng nào có khóa k xuất hiện trong danh sách, thì NIL được trả về. Với danh sách nối kết trong Hình 11.3(a), lệnh gọi LIST-SEARCH (L, 4) trả về một biến trỏ đến thành phần thứ ba, và lệnh gọi LIST-SEARCH(L, 7) trả về NIL.



Hình 11.3 (a) Một danh sách nối kết đôi L biểu thị tập hợp động {1, 4, 9, 16}. Mỗi thành phần trong danh sách là một đối tượng có các trường cho khóa và các biến trỏ (được nêu rõ bằng các mũi tên) đến các đối tượng trước đó và kế tiếp. Trường *next* của đuôi và trường *prev* của đầu là NIL, được chỉ rõ bằng một dấu sổ chéo. Thuộc tính *head* [L] trỏ đến đầu. **(b)** Theo tiến trình thi hành của LIST-INSERT(L, x), ở đó $key[x] = 25$, danh sách nối kết có một đối tượng mới có khóa 25 làm đầu mới. Đối tượng mới này trỏ đến đầu cũ có khóa 9. **(c)** Kết quả của lệnh gọi LIST-DELETE(L, x) tiếp theo, ở đó x trỏ đến đối tượng có khóa 4.

```
LIST-SEARCH(L, k)
x ← head [L]
while x ≠ NIL và key[x] ≠ k
        do x ← next[x]
```

```
return x
```

Để tìm trong một danh sách n đối tượng, thủ tục LIST-SEARCH bỏ ra $\Theta(n)$ thời gian trong trường hợp xấu nhất, bởi nó có thể phải tìm trong nguyên cả danh sách.

Chèn vào một danh sách nối kết

Cho một thành phần x có trường *key* đã được ấn định, thủ tục LIST-INSERT "loại" [splices] x lên trên đầu danh sách nối kết, như đã nêu trong Hình 11.3(**b**).

```
LIST-INSERT(L, x)•
next[x] ← head[L]
if head [L] ≠ NIL
    then prev[head[L]] ← x
head[L] ← x
prev[x] ← NIL
```

Thời gian thực hiện của LIST-INSERT trên một danh sách n thành phần là $O(1)$.

Xóa ra khỏi danh sách nối kết

Thủ tục LIST-DELETE gỡ bỏ một thành phần x ra khỏi một danh sách nối kết L. Nó phải được gán một biến trỏ đến x, và sau đó nó "loại" x ra khỏi danh sách bằng cách cập nhật các biến trỏ. Nếu muốn xóa một thành phần có một khóa đã cho, trước tiên ta phải gọi LIST-SEARCH để truy lục một biến trỏ đến thành phần đó.

```
LIST-DELETE(L,x)
if prev[x] ≠ NIL
    then next[prev[x]] ← next[x]
    else head[L] ← next[x]
if next[x] ≠ NIL
    then prev[next[x]] ← prev[x]
```

Hình 11.3(**c**) nêu cách xóa một thành phần ra khỏi một danh sách nối kết. LIST-DELETE chạy trong $O(1)$ thời gian, nhưng nếu ta muốn xóa một thành phần có một khóa đã cho, $\Theta(n)$ thời gian là bắt buộc trong trường hợp xấu nhất bởi trước tiên ta phải gọi LIST-SEARCH.

Các cờ hiệu

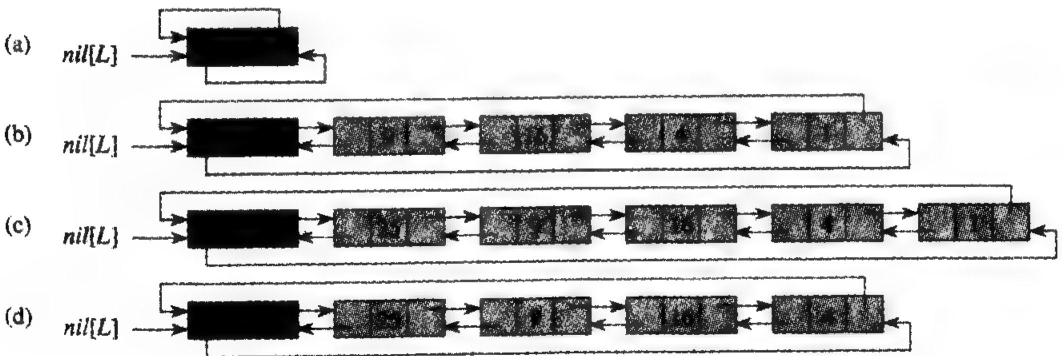
Mã của LIST-DELETE sẽ đơn giản hơn nếu ta có thể bỏ qua các điều

kiện cận tại đầu và đuôi của danh sách.

```
LIST-DELETES (L, x)
next[prev[x]] ← next[x]
prev[next[x]] ← prev[x]
```

Cờ hiệu [sentinel] là một đối tượng giả cho phép ta rút gọn các điều kiện cận. Ví dụ, giả sử ta cung cấp với danh sách *L* một đối tượng *nil* [*L*] biểu diễn NIL nhưng có tất cả các trường của các thành phần danh sách khác. Bất kỳ đâu ta có một tham chiếu đến NIL trong mã danh sách, ta thay nó bằng một tham chiếu đến cờ hiệu *nil*[*L*]. Như đã nêu trong Hình 11.4, điều này chuyển một danh sách nối kết đôi bình thường thành một danh sách vòng, với cờ hiệu *nil* [*L*] nằm giữa đầu và đuôi; trường *next*[*nil*[*L*]] trỏ đến đầu danh sách, và *prev*[*nil* [*L*]] trỏ đến đuôi. Cũng vậy, cả trường *next* của đuôi lẫn trường *prev* của đầu đều trỏ đến *nil*[*L*]. Bởi *next*[*nil*[*L*]] trỏ đến đầu, nên ta có thể loại bỏ thuộc tính *head* [*L*], thay các tham chiếu đến nó bằng các tham chiếu đến *next*[*nil*[*L*]]. Một danh sách trống chỉ bao gồm cờ hiệu, bởi cả hai *next*[*nil*[*L*]] lẫn *prev*[*nil*[*L*]] đều có thể được ấn định là *nil*[*L*].

Mã của LIST-SEARCH vẫn giống như trước, nhưng có các tham chiếu đến NIL và *head*[*L*] đã thay đổi như được chỉ định trên đây.



Hình 11.4 Một danh sách nối kết *L* sử dụng một cờ hiệu *nil*[*L*] (tô đen) là danh sách nối kết đôi bình thường được chuyển thành một danh sách vòng có *nil*[*L*] xuất hiện giữa đầu và đuôi. Thuộc tính *head* [*L*] không còn cần thiết, bởi ta có thể truy cập đầu danh sách theo *next*[*nil*[*L*]]. **(a)** Một danh sách trống. **(b)** Danh sách nối kết từ Hình 11.3**(a)**, có khóa 9 tại đầu và khóa 1 tại đuôi. **(c)** Danh sách sau khi thi hành LIST-INSERT'(*L*,*x*), ở đó *key*[*x*] = 25. Đối tượng mới trở thành đầu danh sách. **(d)** Danh sách sau khi xóa đối tượng có khóa 1. Đuôi mới là đối tượng có khóa 4.

```
LIST-SEARCH'(L, k)
x ← next[nil[L]]
while x ≠ nil [L] và key[x] ≠ k
      do x ← next[x]
return x
```

Ta dùng thủ tục LIST-DELETE' hai dòng để xóa một thành phần ra khỏi danh sách. Dùng thủ tục dưới đây để chèn một thành phần vào danh sách.

```
LIST-INSERT' (L, x)
next[x] ← next[nil [L]]
prev[next[nil[L]]] ← x
next[nil[L]] ← x
prev[x] ← nil [L]
```

Hình 11.4 nêu các hiệu ứng của LIST-INSERT' và LIST-DELETE' trên một danh sách mẫu.

Các cờ hiệu hiếm khi rút gọn các biên thời gian tiệm cận của các phép toán cấu trúc dữ liệu, song có thể rút gọn các thừa số bất biến. Lợi ích của việc dùng các cờ hiệu trong các vòng lặp thường là sự minh bạch của mã thay vì tốc độ; ví dụ, mã của danh sách nối kết được đơn giản hóa nhờ dùng các cờ hiệu, nhưng ta chỉ tiết kiệm $O(1)$ thời gian trong các thủ tục LIST-INSERT' và LIST-DELETE'. Tuy nhiên, trong các tình huống khác, việc dùng các cờ hiệu có thể giúp thắt chặt mã trong một vòng lặp, nhờ đó rút gọn hệ số, giả sử, n hoặc n^2 trong thời gian thực hiện.

Không nên dùng các cờ hiệu một cách cẩu thả. Nếu có nhiều danh sách nhỏ, kho lưu trữ dôi mà các cờ hiệu của chúng sử dụng có thể làm lãng phí một lượng bộ nhớ đáng kể. Trong sách này, ta chỉ dùng các cờ hiệu khi chúng thực sự rút gọn mã.

Bài tập

11.2-1

Có thể thực thi phép toán tập hợp động INSERT trên danh sách nối kết đơn trong $O(1)$ thời gian không? Còn DELETE thì sao?

11.2-2

Thực thi một ngăn xếp dùng danh sách nối kết đơn *L*. Các phép toán PUSH và POP vẫn phải mất $O(1)$ thời gian.

11.2-3

Thực thi một hàng đợi theo một danh sách nối kết đơn *L*. Các phép toán ENQUEUE và DEQUEUE vẫn phải mất $O(1)$ thời gian.

11.2-4

Thực thi các phép toán từ điển INSERT, DELETE, và SEARCH dùng các danh sách vòng, nối kết đơn. Nêu các thời gian thực hiện của các thủ tục?

11.2-5

Phép toán tập hợp động UNION lấy hai tập hợp rời S_1 và S_2 làm đầu vào, và trả về một tập hợp $S = S_1 \cup S_2$ bao gồm tất cả các thành phần của S_1 và S_2. Các tập hợp S_1 và S_2 thường bị hủy bởi phép toán. Nêu cách hỗ trợ UNION trong $O(1)$ thời gian dùng một cấu trúc dữ liệu danh sách thích hợp.

11.2-6

Viết một thủ tục trộn hai danh sách được sắp xếp, nối kết đơn, thành một danh sách sắp xếp, nối kết đơn, mà không dùng các cờ hiệu. Sau đó, viết một thủ tục tương tự dùng một cờ hiệu có khóa ∞ để đánh dấu cuối mỗi danh sách. So sánh tính đơn giản của mã dùng cho hai thủ tục.

11.2-7

Nêu một thủ tục phi đệ quy $\Theta(n)$-thời gian đảo ngược một danh sách nối kết đơn gồm *n* thành phần. Thủ tục không được dùng nhiều hơn kho lưu trữ bất biến vượt quá nhu cầu cần thiết của chính danh sách.

11.2-8 *

Giải thích cách thực thi các danh sách nối kết đôi chỉ dùng một giá trị biến trỏ *np*[*x*] cho mỗi mục thay vì hai (*next* và *prev*) như bình thường. Giả sử tất cả các giá trị chỉ số đều có thể được diễn dịch dưới dạng *k*-bit số nguyên, và định nghĩa *np*[*x*] là *np*[*x*] = *next*[*x*] XOR *prev*[*x*], "exclusive-or" *k*-bit của *next*[*x*] và *prev*[*x*]. (Giá trị NIL được biểu diễn bởi 0.) Đừng quên mô tả phải dùng những thông tin nào để truy cập đầu danh sách. Nêu cách thực thi các phép toán SEARCH, INSERT, và DELETE trên một danh sách như vậy. Ngoài ra, nêu cách nghịch đảo một danh sách như vậy trong $O(1)$ thời gian.

11.3 Thực thi các biến trỏ và các đối tượng

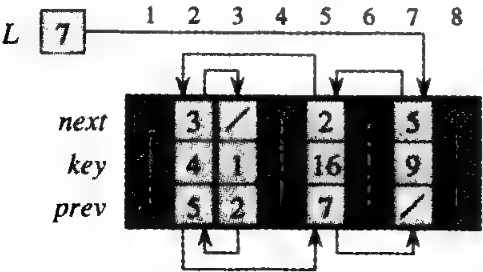
Làm sao để thực thi các biến trỏ và các đối tượng trong các ngôn ngữ không cung cấp chúng, như Fortran? Đoạn này sẽ giải thích hai cách thực thi các cấu trúc dữ liệu nối kết mà không có một kiểu dữ liệu biến trỏ tường minh. Ta sẽ tổng hợp các đối tượng và các biến trỏ từ các mảng và các chỉ số mảng.

Một phần biểu diễn đa mảng của các đối tượng

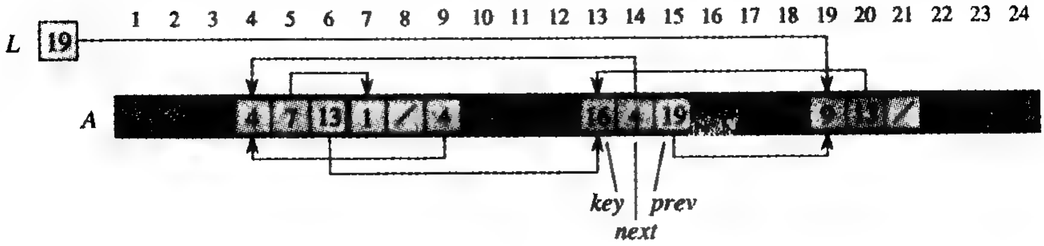
Để biểu diễn một tập hợp các đối tượng có cùng các trường, ta có thể dùng một mảng cho mỗi trường. Để lấy ví dụ, Hình 11.5 nêu cách thực thi danh sách nối kết của Hình 11.3 (a) với ba mảng. Mảng *key* lưu giữ các giá trị của các khóa hiện nằm trong tập hợp động, và các biến trỏ được lưu trữ trong các mảng *next* và *prev*. Với một chỉ số mảng đã cho *x*, *key*[*x*], *next*[*x*], và *prev*[*x*] biểu diễn một đối tượng trong danh sách nối kết. Theo phép diễn dịch này, một biến trỏ *x* đơn giản là một chỉ số chung nhập vào các mảng *key, next,* và *prev*.

Trong Hình 11.3(a), đối tượng có khóa 4 theo sau đối tượng có khóa 16 trong danh sách nối kết. Trong Hình 11.5, khóa 4 xuất hiện trong *key*[2], và khóa 16 xuất hiện trong *key*[5], do đó ta có *next*[5] = 2 và *prev*[2] = 5. Mặc dù hằng NIL xuất hiện trong trường *next* của đuôi và trường *prev* của đầu, song ta thường dùng một số nguyên (như 0 hoặc -1) không thể biểu diễn một chỉ số thực tế vào các mảng. Một biến *L* lưu giữ chỉ số của đầu danh sách.

Trong mã giả, ta đã dùng các dấu ngoặc vuông để thể hiện cả phép lập chỉ số một mảng và phép chọn lọc một trường (thuộc tính) của một đối tượng. Cả hai cách, các ý nghĩa của *key*[*x*], *next*[*x*], và *prev*[*x*] đều nhất quán với thông lệ thực thi.



Hình 11.5 Danh sách nối kết của Hình 11.3(a) được biểu diễn bởi các mảng *key, next,* và *prev*. Mỗi lát dọc của các mảng biểu diễn một đối tượng đơn lẻ. Các biến trỏ trữ sẵn tương ứng với các chỉ số mảng nêu trên cùng; các mũi tên nêu cách diễn dịch chúng. Các vị trí đối tượng được tô bóng sáng chứa các thành phần danh sách. Biến *L* lưu giữ chỉ số của đầu.



Hình 11.6 Danh sách nối kết của các Hình 11.3(a) và 11.5 được biểu diễn trong một mảng đơn lẻ *A*. Mỗi thành phần danh sách là một đối tượng choán một mảng con tiếp cận có chiều dài 3 trong mảng. Ba trường *key*, *next*, và *prev* tương ứng với các độ dịch vị 0, 1, và 2, theo thứ tự nêu trên. Một biến trỏ đến một đối tượng là một chỉ số của thành phần đầu tiên của đối tượng đó. Các đối tượng chứa các thành phần danh sách được tô bóng sáng, và các mũi tên nếu cách sắp xếp thứ tự của danh sách.

Phần biểu diễn mảng đơn của các đối tượng

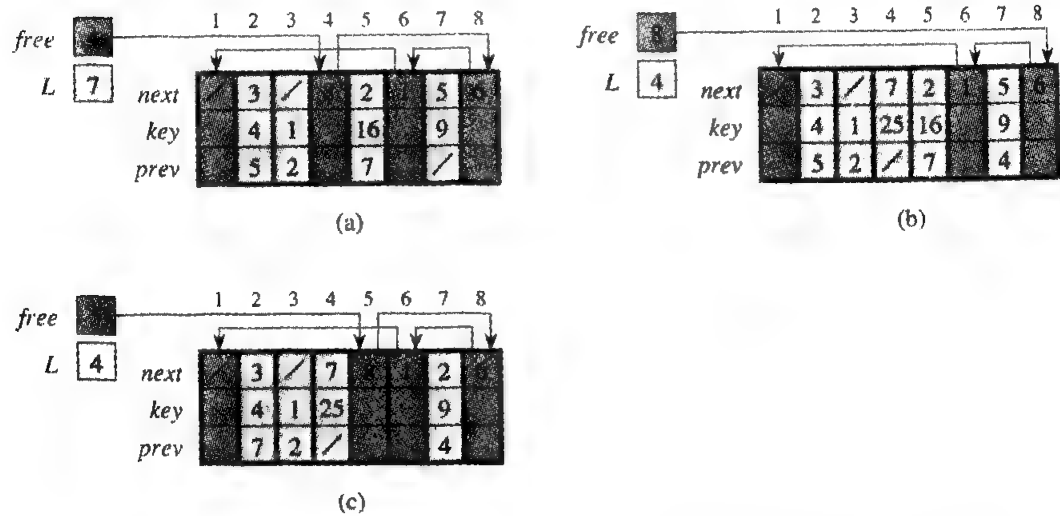
Các từ trong một bộ nhớ máy tính thường được định địa chỉ bởi các số nguyên từ 0 đến M - 1, ở đó M là một số nguyên lớn thích hợp. Trong nhiều ngôn ngữ lập trình, một đối tượng choán một loạt các vị trí tiếp cận trong bộ nhớ máy tính. Một biến trỏ chẳng qua là địa chỉ của vị trí bộ nhớ đầu tiên của đối tượng đó, và có thể lập chỉ mục các vị trí bộ nhớ khác trong đối tượng bằng cách cộng một độ dịch vị đến biến trỏ.

Có thể dùng cùng chiến lược để thực thi các đối tượng trong các môi trường lập trình không cung cấp các kiểu dữ liệu biến trỏ tường minh. Ví dụ, Hình 11.6 nêu cách dùng một mảng *A* đơn lẻ để lưu trữ danh sách nối kết từ các Hình 11.3(a) và 11.5. Một đối tượng choán một mảng con tiếp cận $A[j..k]$. Mỗi trường của đối tượng tương ứng với một độ dịch vị trong miền giá trị từ 0 đến $k - j$, và một biến trỏ đến đối tượng là chỉ số j. Trong Hình 11.6, các độ dịch vị tương ứng với *key*, *next*, và *prev* là 0, 1, và 2, theo thứ tự nêu trên. Để đọc giá trị của $prev[i]$, căn cứ vào một biến trỏ i, ta cộng giá trị i của biến trỏ đến độ dịch vị 2, như vậy là $A[i + 2]$.

Phần biểu diễn mảng đơn tỏ ra linh hoạt ở chỗ nó cho phép lưu trữ các đối tượng có các chiều dài khác nhau trong cùng một mảng. Bài toán quản lý một tập hợp các đối tượng tạp chủng như vậy sẽ khó hơn bài toán quản lý một tập hợp thuần chủng, ở đó tất cả các đối tượng đều có các trường giống nhau. Bởi hầu hết các cấu trúc dữ liệu mà ta sẽ xem xét đều bao gồm các thành phần thuần chủng, nên ta chỉ cần sử dụng phần biểu diễn đa mảng của các đối tượng.

Phân bổ và giải phóng các đối tượng

Để chèn một khóa vào một tập hợp động được biểu diễn bởi một danh sách nối kết đôi, ta phải phân bổ một biến trỏ đến một đối tượng hiện còn rảnh trong phần biểu diễn danh sách nối kết. Như vậy, ta nên quản lý kho lưu trữ các đối tượng hiện chưa dùng trong phần biểu diễn danh sách nối kết sao cho có thể phân bổ một đối tượng. Trong vài hệ thống, một ***trình gom rác*** [garbage collector] sẽ chịu trách nhiệm xác định những đối tượng nào còn rảnh. Tuy nhiên, có nhiều ứng dụng đủ đơn giản để có thể gánh luôn trách nhiệm trả về một đối tượng còn rảnh cho một trình quản lý kho lưu trữ. Giờ đây ta sẽ khảo sát bài toán phân bổ và giải phóng (hoặc thôi phân bổ) các đối tượng thuần chủng, dùng ví dụ về một danh sách nối kết đôi được biểu diễn bởi nhiều mảng.



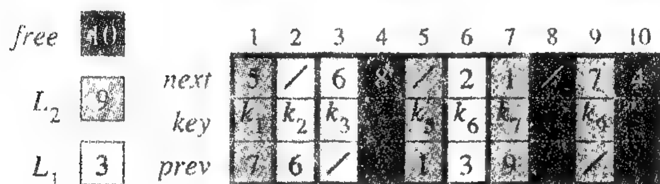
Hình 11.7 Hiệu ứng của các thủ tục ALLOCATE-OBJECT và FREE-OBJECT. **(a)** Danh sách của Hình 11.5 (tô bóng sáng) và một danh sách rảnh (tô bóng đậm). Các mũi tên nêu cấu trúc danh sách rảnh. **(b)** Kết quả của việc gọi ALLOCATE-OBJECT() (trả về chỉ số 4), ấn định *key*[4] là 25, và gọi LIST-INSERT(*L*, 4). Đầu danh sách rảnh mới là đối tượng 8, đã từng là *next*[4] trên danh sách rảnh. **(c)** Sau khi thi hành LIST-DELETE(*L*, 5), ta gọi FREE-OBJECT(5). Đối tượng 5 trở thành đầu danh sách rảnh mới, với đối tượng 8 theo sau nó trên danh sách rảnh.

Giả sử, các mảng trong phần biểu diễn đa mảng có chiều dài m và vào một lúc nào đó tập hợp động chứa $n \leq m$ thành phần. Như vậy, n đối tượng biểu diễn các thành phần hiện nằm trong tập hợp động, và $m - n$ đối tượng còn lại là ***rảnh***; các đối tượng rảnh có thể được dùng để biểu diễn các thành phần được chèn vào tập hợp động trong tương lai.

Ta giữ các đối tượng rảnh trong một danh sách nối kết đơn, mà ta gọi

là ***danh sách rảnh*** [free list]. Danh sách rảnh chỉ sử dụng mảng kế tiếp, lưu trữ các biến trỏ kế tiếp trong danh sách. Đầu danh sách rảnh được lưu giữ trong biến toàn cục *free*. Khi tập hợp động được biểu diễn bởi danh sách nối kết L không trống, danh sách rảnh có thể được đan kết với danh sách L, như đã nêu trong Hình 11.7. Lưu ý, mỗi đối tượng trong phần biểu diễn sẽ hoặc nằm trong danh sách L hoặc trong danh sách rảnh, chứ không phải trong cả hai.

Danh sách rảnh là một ngăn xếp: đối tượng kế tiếp được phân bổ là đối tượng mới được giải phóng. Có thể dùng một thực thi danh sách đối với các phép toán ngăn xếp PUSH và POP để thực thi các thủ tục phân bổ và giải phóng các đối tượng, theo thứ tự nêu trên.



Hình 11.8 Hai danh sách nối kết, L_1, (tô bóng sáng) và L_2 (tô bóng đậm), và một danh sách rảnh (tô sẫm) đan quyện.

Ta mặc nhận rằng biến toàn cục *free* được dùng trong các thủ tục dưới đây trỏ đến thành phần đầu tiên của danh sách rảnh.

ALLOCATE-OBJECT()

```
if free = NIL
      then error "out of space"
      else x ← free
            free ← next[x]
            return x
```

FREE-OBJECT(x)

```
next[x] ← free
free ← x
```

Thoạt đầu, danh sách rảnh chứa tất cả n đối tượng chưa phân bổ. Khi danh sách rảnh đã cạn kiệt, thủ tục ALLOCATE-OBJECT phát tín hiệu một lỗi. Ta thường dùng một danh sách rảnh đơn lẻ để phục vụ cho vài danh sách nối kết. Hình 11.8 nêu ba danh sách nối kết và một danh sách rảnh đan quyện qua các mảng *key*, *next*, và *prev*.

Hai thủ tục chạy trong $O(1)$ thời gian, khiến chúng khá thực tiễn. Có thể sửa đổi để chúng làm việc với bất kỳ tập hợp đối tượng thuần chủng

nào bằng cách cho phép bất kỳ trong số các trường trong đối tượng tác động như một trường *next* trong danh sách rảnh.

Bài tập

11.3-1

Vẽ một bức ảnh dãy <13, 4, 8,19, 5, 11> được lưu trữ dưới dạng một danh sách nối kết đôi dùng phần biểu diễn đa mảng. Cũng làm thế đối với phần biểu diễn mảng đơn.

11.3-2

Viết các thủ tục ALLOCATE-OBJECT và FREE-OBJECT cho một tập hợp các đối tượng thuần chủng được thực thi bởi phần biểu diễn mảng đơn.

11.3 3

Tại sao ta không cần ấn định hoặc chỉnh lại các trường *prev* của các đối tượng trong khi thực thi các thủ tục ALLOCATE-OBJECT và FREE-OBJECT?

11.3-4

Nói chung là thỏa đáng khi nén tất cả các thành phần của một danh sách nối kết đôi trong kho lưu trữ, dùng, chẳng hạn, *m* vị trí chỉ mục đầu tiên trong phần biểu diễn đa mảng. (Đây là trường hợp trong một môi trường điện toán bộ nhớ ảo, có phân trang.) Giải thích cách thực thi các thủ tục ALLOCATE-OBJECT và FREE-OBJECT sao cho phần biểu diễn nén gọn. Giả sử không có các biến trỏ đến các thành phần của danh sách nối kết bên ngoài danh sách đó. (*Mách nước:* Dùng phép thực thi mảng của một ngăn xếp.)

11.3-5

Cho *L* là một danh sách nối kết đôi có chiều dài *m* được lưu trữ trong các mảng *key*, *prev*, và *next* có chiều dài *n*. Giả sử các mảng này được quản lý bởi các thủ tục ALLOCATE-OBJECT và FREE-OBJECT lưu giữ một danh sách rảnh nối kết đôi *F*. Giả sử thêm rằng trong số *n* mục, có chính xác *m* nằm trên danh sách *L* và *n* - *m* nằm trên danh sách rảnh. Viết một thủ tục COMPACTIFY-LIST(*L*, *F*) mà, căn cứ vào danh sách *L* và danh sách rảnh *F*, nó dời các mục trong *L* sao cho chúng choán các vị trí mảng 1, 2,..., *m* và điều chỉnh danh sách rảnh *F* sao cho nó vẫn giữ đúng đắn, choán các vị trí mảng *m* + 1, *m* + 2,..., *n*. Thời gian thực hiện của thủ tục phải là $\Theta(m)$, và nó chỉ dùng một lượng không gian phụ trội bất biến. Nêu một đối số kỹ lưỡng cho tính đúng đắn của thủ tục.

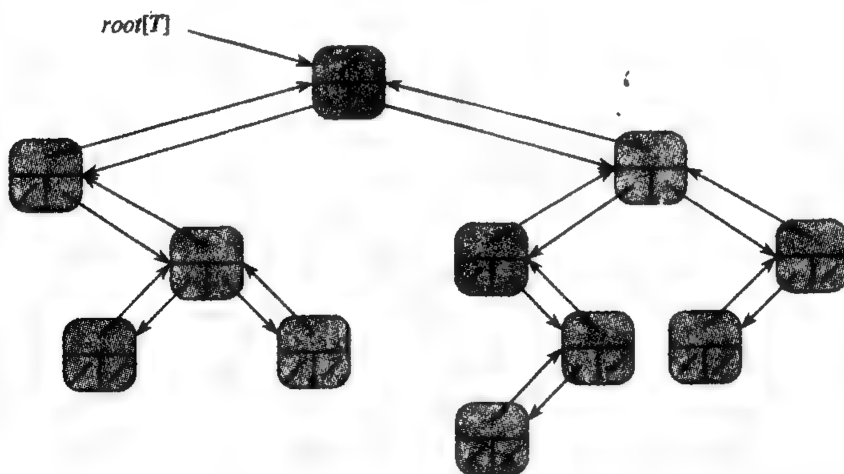
11.4 Biểu diễn các cây có gốc

Các phương pháp để biểu thị các danh sách đã cho trong đoạn trên đây mở rộng ra bất kỳ cấu trúc dữ liệu thuần chủng nào. Đoạn này sẽ xem xét cụ thể bài toán biểu thị các cây có gốc bằng các cấu trúc dữ liệu nối kết. Trước tiên, ta xem xét các cây nhị phân, rồi trình bày một phương pháp cho các cây có gốc ở đó các nút có thể có một số các con tùy ý.

Ta biểu diễn mỗi nút của một cây bằng một đối tượng. Cũng như các danh sách nối kết, ta mặc nhận rằng mỗi nút chứa một trường *key*. Các trường đáng quan tâm còn lại là các biến trỏ đến các nút khác, và chúng thay đổi theo kiểu cây.

Các cây nhị phân

Như đã nêu trong Hình 11.9, ta dùng các trường *p*, *left*, và *right* để lưu trữ các biến trỏ đến cha, con trái, và con phải của mỗi nút trong một cây nhị phân *T*. Nếu *p*[*x*] = NIL, thì *x* là gốc. Nếu nút *x* không có con trái, thì *left*[*x*] = NIL, và với con phải cũng tương tự. Thuộc tính *root*[*T*] trỏ đến gốc của nguyên cả cây *T*. Nếu *root*[*T*] = NIL, thì cây trống.



Hình 11.9 Phần biểu diễn của một cây nhị phân *T*. Mỗi nút *x* có các trường *p*[*x*] (trên đầu), *left*[*x*] (dưới trái), và *right*[*x*] (dưới phải). Các trường *key* không được nêu.

Các cây có gốc với tiến trình rẽ nhánh không giới hạn

Có thể mở rộng lược đồ biểu thị một cây nhị phân đến bất kỳ lớp cây nào ở đó số lượng các con của mỗi nút tối đa là một hằng *k*: ta thay các trường *left* và *right* bằng $child_1$, $child_2$,..., $child_k$. Lược đồ này không còn làm việc khi số lượng các con của một nút không giới hạn, bởi ta không

biết bao nhiêu trường (mảng trong phần biểu diễn đa mảng) để phân bổ trước. Hơn nữa, cho dù số lượng các con k được định cận bởi một hằng lớn nhưng hầu hết các nút có một lượng nhỏ các con, ta có thể lãng phí khá nhiều bộ nhớ.

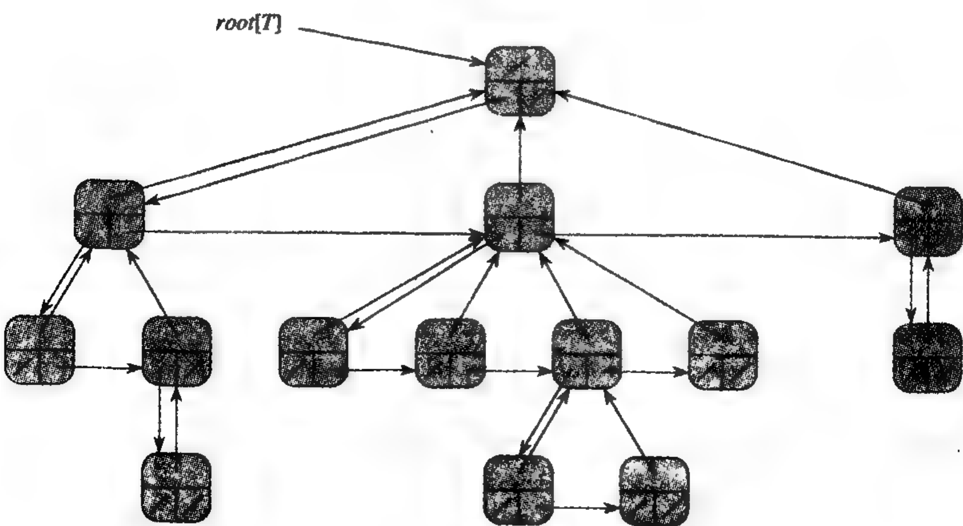
May thay, có một lược đồ thông minh hơn để dùng các cây nhị phân biểu diễn các cây có các số lượng con tùy ý. Ưu điểm của nó là chỉ dùng $O(n)$ không gian với bất kỳ cây có gốc n-nút nào. Hình 11.10 có nêu ***phần biểu diễn left-child, right-sibling***. Cũng như trước, mỗi nút chứa một biến trỏ cha p, và *root*[T] trỏ đến gốc của cây T. Tuy nhiên, thay vì để một biến trỏ đến từng con của nó, mỗi nút x chỉ có hai biến trỏ:

1. *left-child* [x] trỏ đến con mút trái của nút x, và
2. *right-sibling*[x] trỏ đến anh em song sinh của x ngay bên phải.

Nếu nút x không có các con, thì *left-child* [x] = NIL, và nếu nút x là con mút phải của cha của nó, thì *right-sibling*[x] = NIL.

Các phần biểu diễn cây khác

Đôi lúc ta biểu diễn các cây có gốc theo các cách khác. Ví dụ, trong Chương 7, ta đã biểu diễn một đống [heap], dựa trên một cây nhị phân hoàn chỉnh, bởi một mảng đơn cộng với một chỉ số. Các cây xuất hiện trong Chương 22 chỉ được băng ngang về phía gốc, do đó chỉ các biến trỏ cha là hiện diện; không có các biến trỏ đến các con. Có thể có nhiều lược đồ khác. Lược đồ nào là tốt nhất, điều đó tùy thuộc vào ứng dụng.



Hình 11.10 Phần biểu diễn con-trái, anh em song sinh-phải của một cây T. Mỗi nút x có các trường $p[x]$ (trên đầu), *left-child*[x] (dưới trái), và *right-sibling*[x] (dưới phải). Các khóa không được nêu.

Bài tập

11.4-1

Vẽ cây nhị phân có gốc tại chỉ số 6 được biểu thị bởi các trường sau đây.

chỉ số	*key*	*left*	*right*
1	12	7	3
2	15	8	NIL
3	4	10	NIL
4	10	5	9
5	2	NIL	NIL
6	18	1	4
7	7	NIL	NIL
8	14	6	2
9	21	NIL	NIL
10	5	NIL	NIL

11.4-2

Viết một thủ tục đệ quy $O(n)$-thời gian mà, căn cứ vào một cây nhị phân n-nút, in ra khóa của mỗi nút trong cây.

11.4-3

Viết một thủ tục phi đệ quy $O(n)$-thời gian mà, căn cứ vào một cây nhị phân n-nút, in ra khóa của mỗi nút trong cây. Dùng một ngăn xếp làm cấu trúc dữ liệu phụ trợ.

11.4-4

Viết một thủ tục $O(n)$-thời gian in tất cả các khóa của một cây có gốc tùy ý với n nút, ở đó cây được lưu trữ dùng phần biểu diễn con-trái, anh em song sinh-phải.

11.4-5 *

Viết một thủ tục phi đệ quy $O(n)$-thời gian mà, căn cứ vào một cây nhị phân n-nút, in ra khóa của mỗi nút. Không dùng trên mức không gian phụ trội bất biến bên ngoài của chính cây và không sửa đổi cây, thậm chí chỉ là tạm thời, trong thủ tục.

11.4-6 *

Phần biểu diễn left-child, right-sibling của một cây có gốc tùy ý sử dụng ba biến trỏ trong mỗi nút: *left-child, right-sibling*, và *parent*. Từ

một nút bất kỳ, ta có thể đạt đến và định danh cha cùng tất cả các con của nút. Nêu cách hoàn thành cùng hiệu ứng mà chỉ dùng hai biến trỏ và một giá trị Bool trong mỗi nút.

Các Bài Toán

11-1 Các phép so sánh giữa các danh sách

Với mỗi trong bốn kiểu danh sách trong bảng dưới đây, đâu là thời gian thực hiện tiệm cận trong trường hợp xấu nhất của mỗi phép toán tập hợp động được nêu?

	nối kết đơn, không sắp xếp thứ tự	nối kết đơn, có sắp xếp thứ tự	nối kết đôi, không sắp xếp thứ tự	nối kết đôi, có sắp xếp thứ tự
SEARCH(L, k)				
INSERT(L, x)				
DELETE(L, x)				
SUCCESSOR(L, x)				
PREDECESSOR(L, x)				
MINIMUM(L)				
MAXIMUM(L)				

11-2 Các đống khả trộn dùng các danh sách nối kết

Một *đống khả trộn* [mergeable heap] hỗ trợ các phép toán sau: MAKE-HEAP (tạo một trống đống khả trộn), INSERT, MINIMUM, EXTRACT-MIN, và UNION. Nêu cách thực thi các đống khả trộn dùng các danh sách nối kết trong từng trường hợp sau đây. Gắng tạo mỗi phép toán càng hiệu quả càng tốt. Phân tích thời gian thực hiện của mỗi phép toán theo dạng kích cỡ của (các) tập hợp động đang được tiến hành.

a. Các danh sách có sắp xếp.

b. Các danh sách không sắp xếp.

c. Các danh sách không sắp xếp, và các tập hợp động (sẽ hợp nhất) nằm tách rời nhau.

11-3 Tìm trong một danh sách nén có sắp xếp

Bài tập 11.3-4 yêu cầu cách có thể duy trì một danh sách n -thành phần ở dạng nén trong n vị trí đầu tiên của một mảng. Ta sẽ mặc nhận

tất cả các khóa đều riêng biệt và danh sách nén cũng được sắp xếp, nghĩa là, $key[i] < key[next[i]]$ với tất cả $i = 1, 2, ..., n$ sao cho $next[i] \neq$ NIL. Dựa trên các giả thiết này, ta dự kiến có thể dùng thuật toán ngẫu nhiên hóa dưới đây để tìm trong danh sách nhanh hơn nhiều so với thời gian tuyến tính.

```
COMPACT-LIST-SEARCH(L, k)
1  i ← head[L]
2  n ← length[L]
3  while i ≠ NIL và key[i] < k
4       do j ← RANDOM (1, n)
5          if key[i] < key[j] và key[j] < k
6             then i ← j
7          i ← next[i]
8          if key[i] = k
9             then return i
10 return NIL
```

Nếu bỏ qua các dòng 4-6 của thủ tục, ta có thuật toán bình thường để tìm trong một danh sách nối kết có sắp xếp, ở đó chỉ số i lần lượt trỏ đến từng vị trí của danh sách. Các dòng 4-6 gắng nhảy tới trước đến một vị trí đã chọn theo ngẫu nhiên j. Một đợt nhảy như vậy sẽ có lợi nếu $key[j]$ lớn hơn $key[i]$ và nhỏ hơn k; trong trường hợp đó, j đánh dấu một vị trí trong danh sách mà i sẽ phải đi qua trong một đợt tìm kiếm danh sách bình thường. Do danh sách ở dạng nén, nên ta biết bất kỳ chọn lựa nào của j giữa 1 và n đều lập chỉ mục một đối tượng nào đó trong danh sách thay vì một khe trên danh sách rảnh.

a. Tại sao ta mặc nhận tất cả các khóa là riêng biệt trong COMPACT-LIST-SEARCH? Chứng tỏ các đợt nhảy ngẫu nhiên không nhất thiết trợ giúp theo tiệm cận khi danh sách chứa các giá trị khóa lặp lại.

Ta có thể phân tích khả năng thực hiện của COMPACT-LIST-SEARCH bằng cách tách tiến trình thi hành của nó thành hai giai đoạn. Trong giai đoạn đầu, ta không để ý đến các tiến độ về phía tìm k mà các dòng 7-9 hoàn thành. Nghĩa là, giai đoạn 1 chỉ bao gồm việc dời tới trong danh sách theo các lần nhảy ngẫu nhiên. Cũng vậy, giai đoạn 2 không để ý đến tiến độ mà các dòng 4-6 hoàn thành, và như vậy nó nó hoạt động giống như đợt tìm kiếm tuyến tính bình thường.

Cho X_t là biến ngẫu nhiên mô tả khoảng cách trong danh sách nối kết (nghĩa là, qua chuỗi các biến trỏ *next*) từ vị trí i đến khóa k mong

muốn sau t lần lặp của giai đoạn 1.

b. Chứng tỏ thời gian thực hiện dự trù của COMPACT-LIST-SEARCH là $O(t + \mathrm{E}[X_t])$ với tất cả $t \geq 0$.

c. Chứng tỏ $\mathrm{E}[X_t] \leq \sum_{r=0}^{n} (1 - r/n)^t$. (*Mách nước*: Dùng phương trình (6.28).)

d. Chứng tỏ $\sum_{r=0}^{n} r^t \leq n^{t+1}/(t + 1)$.

e. Chứng minh $\mathrm{E}[X_t] \leq n/(t + 1)$, và giải thích tại sao công thức này có ý nghĩa trực giác.

f. Chứng tỏ COMPACT-LIST-SEARCH chạy trong $O(\sqrt{n})$ thời gian dự trù.

Ghi chú Chương

Aho, Hopcroft, và Ullman [5] và Knuth [121] là các tham khảo tuyệt vời về các cấu trúc dữ liệu cơ bản. Gonnet [90] cung cấp dữ liệu thực nghiệm về khả năng thực hiện nhiều phép toán cấu trúc dữ liệu.

Ta không rõ nguồn gốc của các ngăn xếp và các hàng đợi dưới dạng các cấu trúc dữ liệu trong khoa học máy tính, bởi các khái niệm tương ứng đã có sẵn trong toán học và các thói quen kinh doanh gốc giấy tờ trước khi các máy tính kỹ thuật số ra đời. Knuth [121] trích dẫn A. M. Turing như là người phát triển các ngăn xếp để nối kết chương trình con vào năm 1947.

Các cấu trúc dữ liệu gốc biến trỏ cũng dường như là một sáng kiến dân gian. Theo Knuth, các biến trỏ hình như đã được dùng trong các máy tính tiên khởi với các bộ nhớ trống. Ngôn ngữ *A*-1 do G. M. Hopper phát triển vào năm 1951 đã biểu diễn các công thức đại số dưới dạng các cây nhị phân. Knuth cho rằng ngôn ngữ IPL-II, được A. Newell, J. C. Shaw, và H. A. Simon phát triển vào năm 1956, đã nân5 ra tầm quan trọng và đẩy mạnh việc dùng các biến trỏ. Ngôn ngữ IPL-III của họ, được phát triển vào năm 1957, có gộp các phép toán ngăn xếp tường minh.

12 Các Bảng ánh số

Nhiều ứng dụng yêu cầu một tập hợp động chỉ hỗ trợ các phép toán từ điển INSERT, SEARCH, và DELETE. Ví dụ, một bộ biên dịch cho một ngôn ngữ máy tính duy trì một bảng ký hiệu, ở đó các khóa của các thành phần là các chuỗi ký tự tùy ý tương ứng với các dấu định danh [identifiers] trong ngôn ngữ. Một bảng ánh số [hash table] là một cấu trúc dữ liệu hiệu quả để thực thi các từ điển. Mặc dù thực tế có thể tiến hành tìm kiếm một thành phần trong một bảng ánh số miễn là tìm kiếm một thành phần trong một danh sách nối kết—$\Theta(n)$ thời gian trong ca xấu nhất—kỹ thuật ánh số thực hiện cực kỳ tốt. Dưới các giả thiết hợp lý, thời gian dự trù để tìm kiếm một thành phần trong một bảng ánh số là $O(1)$.

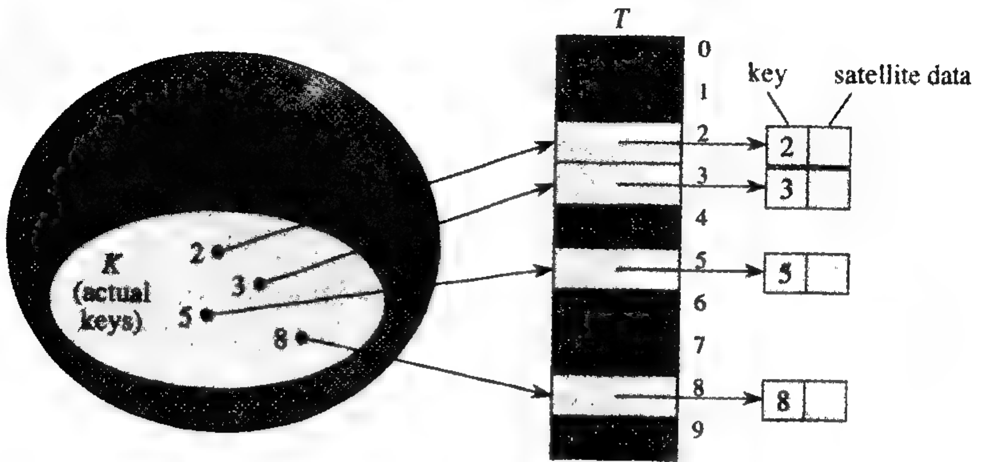
Bảng ánh số là sự tổng quát hóa khái niệm đơn giản hơn của một mảng bình thường. Việc định địa chỉ trực tiếp vào một mảng bình thường vận dụng hiệu quả khả năng của chúng ta để xem xét một vị trí tùy ý trong một mảng trong $O(1)$ thời gian. Đoạn 12.1 đề cập tính năng định địa chỉ trực tiếp trong chi tiết hơn. Việc định địa chỉ trực tiếp is có thể áp dụng khi ta có thể có đủ khả năng to phân bổ a mảng that has một vị trí for mọi khả dĩ khóa.

Khi số lượng khóa được lưu trữ thực tế là tương đối nhỏ so với tổng số các khóa khả dĩ, các bảng ánh số trở thành một phương án thay thế hiệu quả cho tiến trình định địa chỉ trực tiếp một mảng, bởi một bảng ánh số thường sử dụng một mảng có kích cỡ tỷ lệ với số lượng các khóa được lưu trữ thực tế. Thay vì trực tiếp dùng khóa làm một chỉ số mảng, chỉ số mảng được *tính toán* từ khóa đó. Đoạn 12.2 trình bày các ý tưởng chính, và Đoạn 12.3 mô tả cách tính toán các chỉ số mảng từ các khóa nhờ các hàm ánh số. Ta cũng sẽ trình bày và phân tích vài biến thể dựa trên chủ đề biến tấu cơ bản; điểm cốt lõi đó là: ánh số là một kỹ thuật cực kỳ hiệu quả và thực tiễn: các phép toán từ điển cơ bản chỉ yêu cầu $O(1)$ thời gian, tính trung bình.

12.1 Các bảng địa chỉ trực tiếp

Việc định địa chỉ trực tiếp là một kỹ thuật đơn giản làm việc tốt khi

không gian *U* các khóa nhỏ vừa phải. Giả sử, một ứng dụng cần một tập hợp động ở đó mỗi thành phần có một khóa được rút từ không gian $U = \{0, 1, ..., m - 1\}$, ở đó *m* không quá lớn. Ta sẽ mặc nhận không có hai thành phần có chung một khóa.



Hình 12.1 Cách thực thi một tập hợp động theo một bảng địa chỉ trực tiếp *T*. Mỗi khóa trong không gian $U = \{0, 1, ..., 9\}$ tương ứng với một chỉ số trong bảng. Tập hợp $K = \{2, 3, 5, 8\}$ các khóa thực tế sẽ xác định các khe trong bảng chứa các biến trỏ đến các thành phần. Các khe khác, tô bóng đậm, chứa NIL.

Để biểu diễn tập hợp động, ta dùng một mảng, hoặc ***bảng địa chỉ trực tiếp***, $T[0..m-1]$, ở đó mỗi vị trí, hoặc ***khe*** [slot], tương ứng với một khóa trong không gian *U*. Hình 12.1 minh họa cách tiếp cận; khe *k* trỏ đến một thành phần trong tập hợp có khóa *k*. Nếu tập hợp không chứa thành phần nào có khóa *k*, thì $T[k]$ = NIL.

Các phép toán từ điển cũng dễ thực thi.

DIRECT-ADDRESS-SEARCH(*T*, *k*)

  return $T[k]$

DIRECT-ADDRESS-INSERT (*T*, *x*)

  $T[key[x]] \leftarrow x$

DIRECT-ADDRESS-DELETE (*T*, *x*)

  $T[key[x]] \leftarrow$ NIL

Các phép toán này chạy nhanh: chỉ mất $O(1)$ thời gian.

Với vài ứng dụng, các thành phần trong tập hợp động có thể được lưu trữ trong chính bảng địa chỉ trực tiếp. Nghĩa là, thay vì lưu trữ khóa và dữ liệu vệ tinh của một thành phần trong một đối tượng bên ngoài bảng địa chỉ trực tiếp, với một biến trỏ từ một khe trong bảng đến đối tượng, ta có thể lưu trữ đối tượng trong chính khe, như vậy sẽ tiết kiệm được không gian. Hơn nữa, thông thường việc lưu trữ trường *key* của đối tượng là không cần thiết, bởi nếu có chỉ số của một đối tượng trong bảng, ta sẽ có khóa của nó. Tuy nhiên, nếu các khóa không được lưu trữ, ta phải có cách nào đó để biết khe có trống hay không.

Bài tập

12.1-1

Xem xét một tập hợp động S được biểu thị bởi một bảng địa chỉ trực tiếp T có chiều dài m. Mô tả một thủ tục tìm thành phần cực đại của S. Đâu là khả năng thực hiện ca xấu nhất của thủ tục?

12.1-2

Một ***vector bit*** đơn giản là một mảng các bit (các số 0 và 1). Một vector bit có chiều dài m sẽ choán ít chỗ hơn so với một mảng m biến trỏ. Mô tả cách sử dụng một vector bit để biểu diễn một tập hợp động gồm các thành phần riêng biệt không có dữ liệu vệ tinh. Các phép toán từ điển sẽ chạy trong $O(1)$ thời gian.

12.1-3

Đề xuất cách thực thi một bảng địa chỉ trực tiếp ở đó các khóa của các thành phần trữ sẵn không cần phải riêng biệt và các thành phần có thể có dữ liệu vệ tinh. Cả ba phép toán từ điển (INSERT, DELETE, và SEARCH) sẽ chạy trong $O(1)$ thời gian. (Đừng quên DELETE chấp nhận dưới dạng một đối số một biến trỏ đến một đối tượng sẽ được xóa, chứ không phải một khóa.)

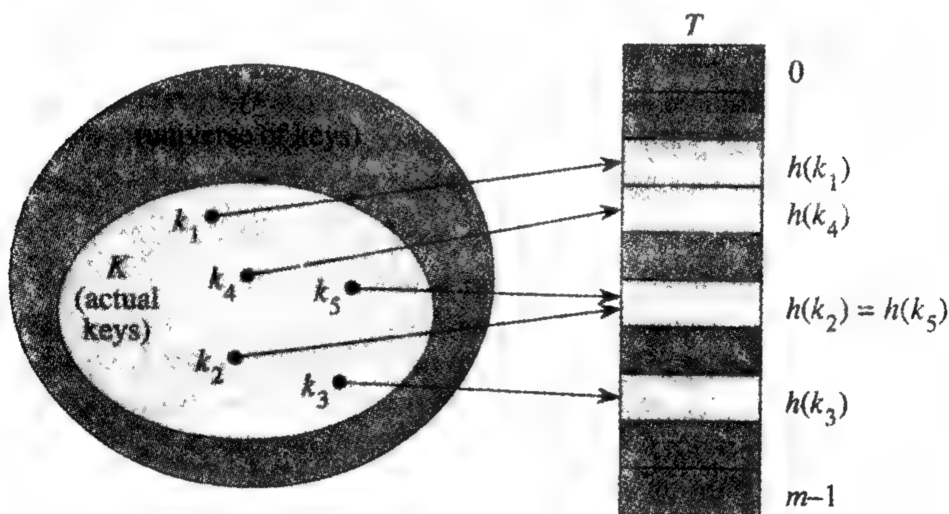
12.1-4

Ta muốn thực thi một từ điển bằng cách dùng tính năng định địa chỉ trực tiếp trên một mảng *khổng lồ*. Ngay từ đầu, các khoản nhập mảng có thể chứa rác, và việc khởi tạo nguyên cả mảng là không thực tiễn bởi kích cỡ của nó. Mô tả một lược đồ để thực thi một từ điển địa chỉ trực tiếp trên một mảng khổng lồ. Mỗi đối tượng trữ sẵn sẽ dùng $O(1)$ không gian; từng phép toán SEARCH, INSERT, và DELETE sẽ chiếm $O(1)$ thời gian; và tiến trình khởi tạo của cấu trúc dữ liệu sẽ mất $O(1)$ thời gian. (*Mách nước:* Dùng một ngăn xếp bổ sung, có kích cỡ là số lượng các khóa thực tế lưu trữ trong từ điển, để giúp xác định xem một khoản nhập đã cho trong mảng khổng lồ có hợp lệ hay không).

12.2 Các bảng ánh số

Sự khó khăn đối với việc định địa chỉ trực tiếp là hiển nhiên: nếu không gian *U* lớn, việc lưu trữ một bảng *T* có kích cỡ $|U|$ có thể không thực tiễn, thậm chí không thể, căn cứ vào bộ nhớ sẵn có trên một máy tính điển hình. Vả lại, tập hợp *K* các khóa *được lưu trữ thực tế* có thể nhỏ tương đối với *U* đến nỗi hầu hết không gian phân bổ cho *T* sẽ bị lãng phí.

Khi tập hợp *K* các khóa lưu trữ trong một từ điển nhỏ hơn nhiều so với không gian *U* tất cả các khóa khả dĩ, một bảng ánh số sẽ yêu cầu ít không gian lưu trữ hơn nhiều so với một bảng địa chỉ trực tiếp. Cụ thể, các yêu cầu kho lưu trữ có thể giảm đến mức $\Theta(|K|)$, cho dù việc tìm kiếm một thành phần trong bảng ánh số vẫn chỉ yêu cầu $O(1)$ thời gian. (Hạn chế duy nhất đó là phạm vi này dành cho *thời gian trung bình*, trong khi đó với tính năng định địa chỉ trực tiếp, nó áp dụng cho *thời gian trường hợp xấu nhất*.)



Hình 12.2 Dùng một hàm ánh số *h* để ánh xạ [map] các khóa theo các khe bảng ánh số. Các khóa k_2 và k_5 ánh xạ theo cùng khe, do đó chúng va chạm.

Với tính năng định địa chỉ trực tiếp, một thành phần có khóa *k* được lưu trữ trong khe *k*. Với kỹ thuật ánh số, thành phần này được lưu trữ trong khe *h*(*k*); nghĩa là, một ***hàm ánh số*** [hash function] *h* được dùng để tính toán khe từ khóa *k*. Ở đây *h* ánh xạ không gian *U* các khóa vào các khe của một *bảng ánh số* $T[0..m-1]$:

$$h: U \rightarrow \{0, 1, ..., m-1\}.$$

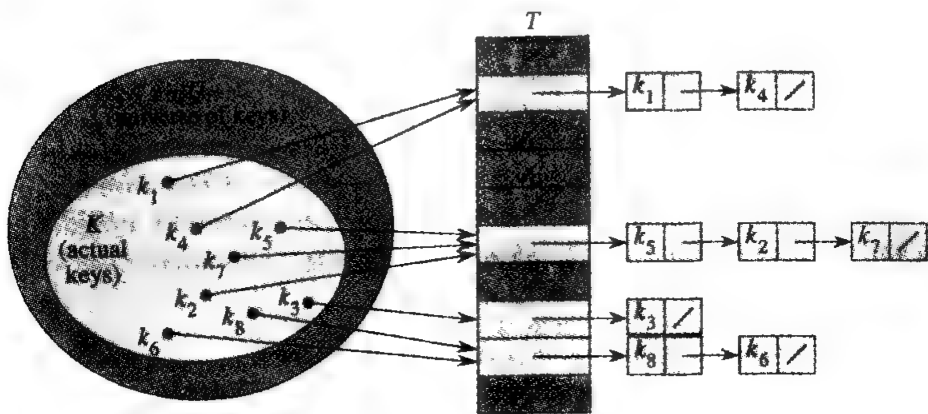
Ta nói rằng một thành phần có khóa *k* sẽ ***ánh số*** [hashes] theo khe

$h(k)$; ta cũng nói rằng $h(k)$ là ***giá trị ánh số*** [hash value] của khóa k. Hình 12.2 minh họa khái niệm căn bản. Cốt lõi của hàm ánh số đó là rút gọn miền các chỉ số mảng cần được điều quản. Thay vì $|U|$ giá trị, ta chỉ cần điều quản m giá trị. Nhờ đó, các yêu cầu kho lưu trữ cũng được thu giảm tương ứng.

Con sâu làm rầu khái niệm đẹp đẽ này đó là: hai khóa có thể ánh số theo cùng khe—một ***va chạm***. May thay, đã có các kỹ thuật hiệu quả để giải quyết sự mâu thuẫn do các va chạm này tạo ra.

Tất nhiên, giải pháp lý tưởng đó là tránh hẳn các va chạm. Ta có thể gắng đạt được mục tiêu này bằng cách chọn một hàm ánh số h thích hợp. Một ý kiến đó là để h xuất hiện "ngẫu nhiên," nhờ đó tránh được các va chạm hoặc chí ít cũng giảm thiểu số lần của chúng. Đúng nghĩa động từ "to hash" [*trong tiếng Anh có nghĩa là băm trộn*__ND], gợi lên hình ảnh băm và trộn ngẫu nhiên, đã nói lên tinh thần của cách tiếp cận này. (Tất nhiên, một hàm ánh số h phải tất định ở chỗ một đầu vào k đã cho phải luôn tạo ra cùng kết xuất $h(k)$.) Tuy nhiên, do $|U| > m$, nên phải có hai khóa có cùng giá trị ánh số; do đó việc tránh hẳn các va chạm là không thể. Như vậy, tuy hàm ánh số ra vẻ "ngẫu nhiên" và được thiết kế kỹ lưỡng có thể giảm thiểu số lần va chạm, song ta vẫn phải có một phương pháp để giải quyết các va chạm tất xảy ra.

Phần còn lại của đoạn này trình bày một kỹ thuật giải quyết va chạm đơn giản nhất, có tên "kết xích" [chaining]. Đoạn 12.4 giới thiệu một phương pháp khác để giải quyết các va chạm, có tên lập địa chỉ mở [open addressing].



Hình 12.3 cách giải quyết va chạm bằng kết xích. Mỗi khe bảng ánh số $T[j]$ chứa một danh sách nối kết bao gồm tất cả các khóa có giá trị ánh số là j. Ví dụ, $h(k_1) = h(k_4)$ và $h(k_5) = h(k_2) = h(k_7)$.

Giải quyết va chạm bằng kết xích

Trong kỹ thuật ***kết xích*** [chaining], ta đặt tất cả các thành phần ánh số theo cùng khe vào trong một danh sách nối kết, như đã nêu trong Hình 12.3. Khe j chứa một biến trỏ đến đầu của danh sách tất cả các thành phần đã lưu trữ ánh số theo j; nếu không có thành phần nào như vậy, khe j chứa NIL.

Các phép toán từ điển trên một bảng ánh số T thường dễ thực thi khi các va chạm được giải quyết bằng kỹ thuật kết xích.

CHAINED-HASH-INSERT(T, x)

chèn x tại đầu danh sách $T[h(key[x])]$

CHAINED-HASH-SEARCH(T, k)

tìm kiếm một thành phần có khóa k trong danh sách $T[h(k)]$

CHAINED-HASH-DELETE(T, x)

xóa x ra khỏi danh sách $T[h(key[x])]$

Thời gian thực hiện trường hợp xấu nhất của phương pháp chèn là $O(1)$. Với tiến trình tìm kiếm, thời gian thực hiện trường hợp xấu nhất sẽ tỷ lệ với chiều dài danh sách; ta sẽ phân tích điều này kỹ hơn dưới đây. Việc xóa một thành phần x có thể hoàn tất trong $O(1)$ thời gian nếu các danh sách được nối kết đôi. (Nếu các danh sách được nối kết đơn, trước tiên ta phải tìm ra x trong danh sách $T[h(key[x])]$, sao cho nối kết *next* của phần tử tiền vị x có thể được ấn định đúng đắn để "loại" x ra; trong trường hợp này, tiến trình xóa và tìm kiếm chủ yếu có cùng thời gian thực hiện.)

Phân tích kỹ thuật ánh số bằng kết xích

Kỹ thuật ánh số bằng kết xích thực hiện tốt tới mức nào? Nói cụ thể, phải mất bao lâu để tìm kiếm một thành phần có một khóa đã cho?

Cho một bảng ánh số T có m khe lưu trữ n thành phần, ta định nghĩa ***hệ số tải*** [load factor] α của T là n/m, nghĩa là, số trung bình các thành phần được lưu trữ trong một xích. Phân tích của chúng ta sẽ theo dạng α; nghĩa là, ta tưởng tượng α đứng yên khi n và m đi đến vô cực. (Lưu ý, α có thể nhỏ hơn, bằng, hoặc lớn hơn 1.)

Cách ứng xử xấu nhất của phương pháp chèn là $O(1)$. Với tiến trình tìm k xấu nhất của kỹ thuật ánh số bằng kết xích thật tồi tệ: tất cả n khóa đều ánh số cùng một khe, tạo ra một danh sách có chiều dài n. Như vậy, thời gian trường hợp xấu nhất để tìm kiếm là $\Theta(n)$ cộng thời gian để tính toán hàm ánh số—không tốt hơn so với việc dùng một danh sách nối kết cho tất cả các thành phần. Rõ ràng, các bảng ánh số không được dùng cho

khả năng thực hiện trường hợp xấu nhất của chúng.

Khả năng thực hiện trung bình của kỹ thuật ánh số tùy thuộc vào mức độ tốt xấu khi hàm ánh số h phân phối tập hợp của các khóa được lưu trữ giữa m khe, tính trung bình. Đoạn 12.3 mô tả các vấn đề này, nhưng trước nút ta mặc nhận rằng mọi thành phần đã cho đều có khả năng như nhau để ánh số vào bất kỳ trong số m khe, độc lập với nơi mà bất kỳ thành phần nào khác đã ánh số theo. Ta gọi điều này là giả thiết của ***kỹ thuật ánh số đều đơn giản*** [simple uniform hashing].

Ta mặc nhận rằng giá trị ánh số $h(k)$ có thể được tính toán trong $O(1)$ thời gian, sao cho thời gian cần thiết để tìm kiếm một thành phần có khóa k tùy thuộc (theo tuyến tính) vào chiều dài của danh sách $T[h(k)]$. Tạm gạt sang một bên $O(1)$ thời gian cần thiết để tính toán hàm ánh số và truy cập khe $h(k)$, ta hãy chú ý đến số thành phần dự trù mà thuật toán tìm kiếm xem xét, nghĩa là, số lượng các thành phần trong danh sách $T[h(k)]$ được kiểm tra để xem các khóa của chúng có bằng với k hay không. Ta sẽ xét hai trường hợp. Trong trường hợp đầu tiên, đợt tìm kiếm không thành công: không có thành phần nào trong bảng có khóa k. Trong trường hợp thứ hai, đợt tìm kiếm thành công với thành phần có khóa k.

Định lý 12.1

Trong một bảng ánh số ở đó các va chạm được giải quyết bằng kỹ thuật kết xích, một đợt tìm kiếm không thành công bình quân sẽ mất một thời gian $\Theta(1+\alpha)$, dưới giả thiết kỹ thuật ánh số đều đơn giản.

Chứng minh Dưới giả thiết kỹ thuật ánh số đều đơn giản, mọi khóa k đều có khả năng như nhau để ánh số theo bất kỳ trong số m khe. Như vậy, thời gian trung bình để tìm kiếm không thành công một khóa k là thời gian trung bình để tìm đến cuối của một trong m danh sách. Chiều dài trung bình của một danh sách như vậy là hệ số tải $\alpha = n/m$. Như vậy, số thành phần dự trù được xem xét trong đợt tìm kiếm không thành công là α, và tổng thời gian cần thiết (kể cả thời gian để tính toán $h(k)$) là $\Theta(1+\alpha)$.

Định lý 12.2

Trong một bảng ánh số ở đó các va chạm được giải quyết bằng kỹ thuật kết xích, một đợt tìm kiếm thành công bình quân sẽ mất một thời gian $\Theta(1+\alpha)$, dưới giả thiết kỹ thuật ánh số đều đơn giản.

Chứng minh Ta mặc nhận rằng khóa đang được tìm kiếm có khả năng như nhau để trở thành bất kỳ trong n khóa được lưu trữ trong bảng. Ta cũng mặc nhận rằng thủ tục CHAINED-HASH-INSERT chèn một thành phần mới tại cuối danh sách thay vì tại đầu. (Qua Bài tập 12.2-3, thời gian tìm kiếm thành công trung bình là như nhau dẫu các thành

phần mới được chèn tại đầu hay tại cuối danh sách.) Số thành phần dự trù được xem xét trong một đợt tìm kiếm thành công là nhiều hơn 1 so với số lượng các thành phần được xem xét khi thành phần tìm thấy được chèn (bởi mọi thành phần mới đều bắt đầu tại cuối danh sách). Do đó, để tìm ra số thành phần dự trù được xem xét, ta lấy trung bình, trên n mục trong bảng, của 1 cộng với chiều dài dự trù của danh sách mà thành phần thứ i được bổ sung. Chiều dài dự kiến của danh sách đó sẽ là $(i - 1) / m$, và như vậy số thành phần dự trù được xem xét trong một đợt tìm kiếm thành công là

$$\begin{aligned}\frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{i-1}{m}\right) &= 1 + \frac{1}{nm}\sum_{i=1}^{n}(i-1)\\ &= 1 + \left(\frac{1}{nm}\right)\left(\frac{(n-1)n}{2}\right)\\ &= 1 + \frac{\alpha}{2} - \frac{1}{2m}\ .\end{aligned}$$

Như vậy, tổng thời gian cần thiết cho một đợt tìm kiếm thành công (kể cả thời gian để tính toán hàm ánh số) là $\Theta(2 + \alpha/2 - 1/2m) = \Theta(1 + \alpha)$.

Kỹ thuật phân tích này có ý nghĩa gì? Nếu số lượng các khe bảng ánh số ít nhất tỷ lệ với số lượng các thành phần trong bảng, ta có $n = O(m)$ và, bởi vậy, $\alpha = n/m = O(m)/m = O(1)$. Như vậy, tính trung bình việc tìm kiếm bỏ ra thời gian bất biến. Bởi tiến trình chèn mất $O(1)$ thời gian trường hợp xấu nhất (xem Bài tập 12.2-3), và tiến trình xóa mất $O(1)$ thời gian trường hợp xấu nhất khi các danh sách được nối kết đôi, tất cả các phép toán từ điển có thể được xác nhận trong $O(1)$ thời gian theo trung bình.

Bài tập

12.2-1

Giả sử ta dùng một hàm ánh số h ngẫu nhiên để ánh số n khóa riêng biệt vào một mảng T có chiều dài m. Đâu là số lượng các va chạm dự trù? Chính xác hơn, đâu là bản số kỳ vọng của $\{(x, y) : h(x) = h(y)\}$?

12.2-2

Chứng minh việc chèn các khóa 5, 28,19, 15, 20, 33, 12, 17, 10 vào một bảng ánh số có các va chạm được giải quyết bởi kỹ thuật kết xích. Cho bảng có 9 khe, và cho hàm ánh số là $h(k) = k \bmod 9$.

12.2-3

Chứng tỏ thời gian dự trù cho một đợt tìm kiếm thành công với kỹ

thuật kết xích là như nhau dẫu các thành phần mới được chèn tại đầu hay tại cuối của một danh sách. (*Mách nước:* Chứng tỏ thời gian tìm kiếm thành công dự trù là như nhau với *bất kỳ* hai kiểu sắp xếp thứ tự nào của một danh sách bất kỳ.)

12.2-4

Giáo sư Marley cho rằng có thể đạt được các thành quả đáng kể về khả năng thực hiện nếu ta sửa đổi lược đồ kết xích sao cho mỗi danh sách đều được giữ ở tình trạng sắp xếp. Việc sửa đổi của giáo sư ảnh hưởng ra sao đến thời gian thực hiện của các đợt tìm kiếm thành công, các đợt tìm kiếm không thành công, các lần chèn, và xóa?

12.2-5

Gợi ý cách phân bổ và thôi phân bổ kho lưu trữ của các thành phần trong chính bảng ánh số bằng cách nối kết tất cả các khe còn rảnh vào một danh sách rảnh. Giả sử một khe có thể lưu trữ một cờ và hoặc một thành phần cộng với một biến trỏ hoặc hai biến trỏ. Tất cả các phép toán từ điển và danh sách rảnh sẽ chạy trong $O(1)$ thời gian dự trù. Danh sách rảnh cần được nối kết đôi hay chỉ cần một danh sách rảnh nối kết đơn là đủ?

12.2-6

Chứng tỏ nếu $|U| > nm$, ta có một tập hợp con U có kích cỡ n bao gồm các khóa mà tất cả đều ánh số theo cùng khe, sao cho thời gian tìm kiếm trường hợp xấu nhất của kỹ thuật ánh số bằng kết xích là $\Theta(n)$.

12.3 Các hàm ánh số

Trong đoạn này, ta đề cập vài vấn đề liên quan đến thiết kế của các hàm ánh số tốt và sau đó trình bày ba lược đồ để tạo chúng: kỹ thuật ánh số bằng phép chia, kỹ thuật ánh số bằng phép nhân, và kỹ thuật ánh số toàn thể [universal hashing].

Điều gì tạo nên một hàm ánh số tốt?

Một hàm ánh số tốt thỏa (xấp xỉ) giả thiết của kỹ thuật ánh số đều đơn giản: mỗi khóa có khả năng như nhau để ánh số theo bất kỳ trong số m khe. Chính thức hơn, ta hãy mặc nhận rằng mỗi khóa được rút ra độc lập từ U theo một phép phân phối xác suất P; nghĩa là, $P(k)$ là xác suất mà k được rút. Như vậy giả thiết của kỹ thuật ánh số đều đơn giản là

$$\sum_{k:h(k)=j} P(k) = \frac{1}{m} \qquad \text{for } j = 0, 1, \ldots, m-1 \qquad (12.1)$$

Đáng tiếc, ta thường không thể kiểm tra điều kiện này, bởi *P* thường chưa biết.

Đôi lúc (hiếm khi) ta biết được phép phân phối *P*. Ví dụ, giả sử các khóa được xem là các số thực ngẫu nhiên *k* được phân phối độc lập và đồng đều trong miền giá trị $0 \le k < 1$. Trong trường hợp này, hàm ánh số

$$h(k) = \lfloor km \rfloor$$

có thể được chứng tỏ là thỏa phương trình (12.1).

Trong thực tế, có thể dùng các kỹ thuật phỏng đoán [heuristic] để tạo một hàm ánh số có khả năng thực hiện tốt. Thông tin định tính về *P* đôi lúc tỏ ra hữu ích trong tiến trình thiết kế này. Ví dụ, hãy xét bảng ký hiệu của một bộ biên dịch, ở đó các khóa là các chuỗi ký tự tùy ý biểu thị cho các dấu định danh [identifiers] trong một chương trình. Nói chung, các ký hiệu có liên quan mật thiết, như pt và pts, thường xảy ra trong cùng chương trình. Một hàm ánh số tốt sẽ giảm thiểu cơ hội các biến thể như vậy ánh số theo cùng khe.

Một cách tiếp cận chung đó là suy ra giá trị ánh số theo cách được dự trù là độc lập với bất kỳ khuôn mẫu nào có thể tồn tại trong dữ liệu. Ví dụ, "phương pháp chia" (xem phần dưới đây) tính toán giá trị ánh số như là số dư khi khóa được chia cho một số nguyên tố định rõ. Trừ phi số nguyên tố đó bằng cách nào đó có liên quan đến các khuôn mẫu trong phép phân phối xác suất *P*, phương pháp này cho ra các kết quả tốt.

Cuối cùng, cần lưu ý có vài ứng dụng của các hàm ánh số có thể yêu cầu các tính chất mạnh hơn so với các tính chất mà kỹ thuật ánh số đều đơn giản cung cấp. Ví dụ, có thể ta muốn các khóa phải "đóng" theo một nghĩa nào đó để cho ra các giá trị ánh số tách xa nhau. (Tính chất này đặc biệt thỏa đáng khi ta đang dùng phương pháp thăm dò tuyến tính, được định nghĩa trong Đoạn 12.4.)

Diễn dịch các khóa dưới dạng các số tự nhiên

Hầu hết các hàm ánh số đều mặc nhận không gian của các khóa là tập hợp N = {0, 1, 2,...} các số tự nhiên. Như vậy, nếu các khóa không phải là các số tự nhiên, ta phải có một cách nào đó để diễn dịch chúng dưới dạng các số tự nhiên. Ví dụ, một khóa là một chuỗi ký tự có thể được diễn dịch dưới dạng một số nguyên được biểu diễn theo cơ số thích hợp. Như vậy, dấu định danh pt có thể được diễn dịch dưới dạng cặp số nguyên thập phân (112, 116), bởi `p` = 112 và `t`= 116 trong bộ ký tự ASCII; vậy, được diễn tả dưới dạng một số nguyên cơ số-128, `pt` trở thành $(112 \cdot 128) + 116 = 14452$. Nói chung trong bất kỳ ứng dụng đã cho nào, ta có thể dễ dàng nghĩ ra một phương pháp đơn giản như vậy

để diễn dịch từng khóa dưới dạng một số tự nhiên (có thể lớn). Trong nội dung dưới đây, ta mặc nhận các khóa là các số tự nhiên.

12.3.1 Phương pháp chia

Trong ***phương pháp chia*** để tạo các hàm ánh số, ta ánh xạ một khóa k vào một trong số m khe bằng cách lấy số dư của k chia cho m. Nghĩa là, hàm ánh số là

$$h(k)=k \bmod m .$$

Ví dụ, nếu bảng ánh số có kích cỡ $m = 12$ và khóa là $k = 100$, thì $h(k) = 4$. Bởi nó chỉ yêu cầu một phép toán chia đơn lẻ, kỹ thuật ánh số bằng phép chia chạy khá nhanh.

Khi dùng phương pháp chia, ta thường tránh một số giá trị nhất định của m. Ví dụ, m không được là một lũy thừa của 2, bởi nếu $m = 2^p$, thì $h(k)$ chỉ là p bit thứ tự thấp nhất của k. Trừ phi tiên nghiệm rằng phép phân phối xác suất trên các khóa khiến tất cả các khuôn mẫu p-bit thứ tự thấp có khả năng như nhau, tốt nhất ta nên để hàm ánh số tùy thuộc vào tất cả các bit của khóa. Phải tránh các lũy thừa của 10 nếu ứng dụng xử lý các số thập phân dưới dạng các khóa, bởi như vậy hàm ánh số không tùy thuộc vào tất cả các chữ số thập phân của k. Cuối cùng, có thể thấy rằng khi $m = 2^p - 1$ và k là một chuỗi ký tự được diễn dịch theo cơ số 2^p, hai chuỗi đồng nhất ngoại trừ sự hoán vị của hai ký tự kề sẽ ánh số theo cùng giá trị.

Các giá trị tốt cho m là các số nguyên tố không quá sát với các lũy thừa chính xác của 2. Ví dụ, giả sử ta muốn phân bổ một bảng ánh số, có các va chạm được giải quyết bằng kết xích, để lưu giữ đại để $n = 2000$ chuỗi ký tự, ở đó một ký tự có 8 bit. Ta không quan tâm xem xét một số trung bình của 3 thành phần trong một đợt tìm kiếm không thành công, do đó ta phân bổ một bảng ánh số có kích cỡ $m = 701$. Sở dĩ con số 701 được chọn bởi vì nó là một số nguyên tố gần $\alpha = 2000/3$ nhưng không gần bất kỳ lũy thừa nào của 2. Xem mỗi khóa k như một số nguyên, hàm ánh số của chúng ta sẽ là

$$h(k) = k \bmod 701 .$$

Để đề phòng, ta có thể kiểm tra mức độ đều mà hàm ánh số này phân phối các tập hợp khóa giữa các khe, ở đó các khóa được chọn từ dữ liệu "thực".

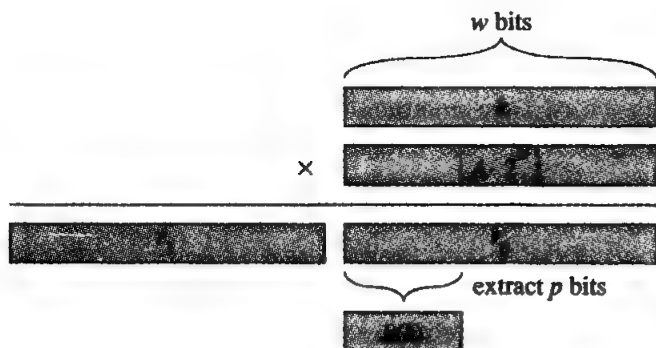
12.3.2 Phương pháp nhân

Phương pháp nhân để tạo các hàm ánh số vận hành theo hai bước. Trước hết, ta nhân khóa k với một hằng A trong miền giá trị $0 < A < 1$ và trích phần phân số của kA. Sau đó, ta nhân giá trị này với m và lấy sàn

của kết quả. Tóm lại, hàm ánh số là

$$h(k) = \lfloor m\,(k\,A \bmod 1) \rfloor ,$$

ở đó "$k\,A$ mod 1" có nghĩa là phần phân số của kA, nghĩa là, $kA - \lfloor kA \rfloor$.



Hình 12.4 Phương pháp nhân của kỹ thuật ánh số. Phần biểu diễn w-bit của khóa k được nhân với giá trị w-bit $[A \,.\, 2^w]$ ở đó $0 < A < 1$ là một hằng thích hợp. p bit cấp cao nhất của nửa w-bit thấp của tích lập thành giá trị ánh số muốn có $h(k)$.

Một ưu điểm của phương pháp nhân đó là giá trị của m là không tới hạn. Ta thường chọn nó là một lũy thừa của 2 - $m = 2^p$ với một số nguyên p—bởi như vậy ta có thể dễ dàng thực thi hàm trên hầu hết các máy tính như sau. Giả sử kích cỡ từ của máy là w bit và k vừa với một từ đơn lẻ. Tham khảo Hình 12.4, trước tiên ta nhân k với số nguyên w -bit $\lfloor A \,.\, 2^w \rfloor$. Kết quả là một giá trị $2w$-bit $r_1 2^w + r_0$, ở đó r_1 là từ thứ tự cao của tích và r_0 là từ thứ tự thấp của tích. Giá trị ánh số p-bit muốn có bao gồm p bit quan trọng nhất của r_0.

Mặc dù phương pháp này làm việc với mọi giá trị của hằng A, song với vài giá trị nó làm việc tốt hơn so với các giá trị khác. Chọn lựa tối ưu tùy thuộc vào các đặc tính của dữ liệu đang được ánh số. Knuth [123] đề cập chi tiết sự chọn lựa của A và gợi ý rằng

$$A \approx (\sqrt{5} - 1)/2 = 0.6180339887... \qquad (12.2)$$

ắt làm việc khá hợp lý.

Để lấy ví dụ, nếu ta có $k = 123456$, $m = 10000$, và A như trong phương trình (12.2), thì

$$\begin{aligned} h(k) &= \lfloor 10000 \,.\, (123456 \,.\, 0.61803... \bmod 1) \rfloor \\ &= \lfloor 10000 \,.\, (76300.0041151... \bmod 1) \rfloor \\ &= \lfloor 10000.0.0041151... \rfloor \\ &= \lfloor 41.151... \rfloor \\ &= 41 . \end{aligned}$$

12.3.3 Kỹ thuật ánh số toàn thể

Nếu một đối phương ác ý chọn các khóa để ánh số, thì hắn có thể chọn n khóa mà tất cả đều ánh số theo cùng khe, cho ra một thời gian truy lục trung bình là $\Theta(n)$. Mọi hàm ánh số cố định đều dễ bị kiểu ứng xử trường hợp xấu nhất này làm tổn thương; cách hiệu quả duy nhất để cải thiện tình huống đó là chọn hàm ánh số *một cách ngẫu nhiên* sao cho *độc lập* với các khóa thực tế sẽ được lưu trữ. Cách tiếp cận này, có tên là ***kỹ thuật ánh số toàn thể*** [universal hashing], cho ra khả năng thực hiện tốt tính theo trung bình, bất chấp đối phương chọn chọn các khóa nào.

Ý tưởng chính nằm sau kỹ thuật ánh số toàn thể đó là lựa hàm ánh số theo ngẫu nhiên vào thời gian thực hiện từ một lớp các hàm được thiết kế cẩn thận. Như trong trường hợp sắp xếp nhanh, việc ngẫu nhiên hóa bảo đảm không có đầu vào đơn lẻ nào sẽ luôn gợi lên cách ứng xử ca xấu nhất. Do sự ngẫu nhiên hóa, thuật toán có thể ứng xử khác nhau trên mỗi lần thi hành, thậm chí với cùng đầu vào. Cách tiếp cận này bảo đảm khả năng thực hiện tốt trường hợp trung bình, bất kể đã cung cấp khóa nào làm đầu vào. Quay lại ví dụ về bảng ký hiệu của bộ biên dịch, ta thấy rằng sự chọn lựa các dấu định danh của lập trình viên giờ đây không thể gây ra khả năng thực hiện kỹ thuật ánh số tồi một cách nhất quán. Khả năng thực hiện tồi chỉ xảy ra nếu bộ biên dịch chọn một hàm ánh số ngẫu nhiên khiến tập hợp các dấu định danh ánh số tồi, nhưng xác suất của trường hợp này không lớn và giống nhau với bất kỳ tập hợp các dấu định danh nào có cùng kích cỡ.

Cho $\mathcal{H}$ là một tập hợp hữu hạn các hàm ánh số ánh xạ một không gian U các khóa đã cho vào miền giá trị $\{0, 1, ..., m - 1\}$. Một tập hợp như vậy được gọi là ***toàn thể*** [universal] nếu với mỗi cặp khóa riêng biệt $x, y \in U$, số lượng hàm ánh số $h \in \mathcal{H}$ qua đó $h(x) = h(y)$ chính xác là $|\mathcal{H}|/m$. Nói cách khác, với một hàm ánh số được chọn ngẫu nhiên từ $\mathcal{H}$, cơ hội của một va chạm giữa x và y khi $x \neq y$ sẽ chính xác là $1/m$, cũng chính xác là cơ hội của một va chạm nếu $h(x)$ và $h(y)$ được chọn ngẫu nhiên từ tập hợp $\{0, 1, ..., m - 1\}$.

Định lý dưới đây cho thấy một lớp toàn thể gồm các hàm ánh số cho ta cách ứng xử trường hợp trung bình tốt.

***Định lý* 12.3**

Nếu h được chọn từ một tập hợp toàn thể các hàm ánh số và được dùng để ánh số n khóa vào một bảng có kích cỡ m, ở đó $n \leq m$, số lần va chạm dự trù liên quan đến một khóa cụ thể x sẽ nhỏ hơn 1.

Chứng minh Với mỗi cặp y, z các khóa riêng biệt, cho c_{yz} là một biến ngẫu nhiên sẽ là 1 nếu $h(y) = h(z)$ (tức là, nếu y và z va chạm dùng h) và

bằng không là 0. Bởi, theo định nghĩa, một cặp khóa đơn lẻ va chạm với xác suất $1/m$, nên ta có

$$E[c_{yz}] = 1/m .$$

Cho C_x là tổng của các va chạm liên quan đến khóa x trong một bảng ánh số T có kích cỡ m chứa n khóa. Phương trình (6.24) cho

$$\begin{aligned} E[C_x] &= \sum_{\substack{y \in T \\ y \neq x}} E[c_{xy}] \\ &= \frac{n-1}{m} . \end{aligned}$$

Bởi $n \leq m$, nên ta có $E[C_x] < 1$.

Nhưng thiết kế một lớp toàn thể các hàm ánh số có dễ không? Nó khá dễ, chỉ cần một ít lý thuyết số sẽ giúp ta chứng minh. Hãy chọn kích cỡ bảng m là số nguyên tố (như trong phương pháp chia). Ta phân tách một khóa x thành $r+1$ byte (tức là, các ký tự, hoặc các chuỗi con nhị phân có chiều rộng cố định), sao cho $x = <x_0, x_1, ..., x_r>$; yêu cầu duy nhất đó là giá trị cực đại của một byte phải nhỏ hơn m. Cho $a = <a_0, a_1, ..., a_r>$ thể hiện một dãy $r + 1$ thành phần được chọn ngẫu nhiên từ tập hợp $\{0,1,..., m - 1\}$. Ta định nghĩa một hàm ánh số tương ứng $h_a \in \mathcal{H}$:

$$h_a(x) = \sum_{i=0}^{r} a_i x_i \bmod m . \tag{12.3}$$

Với định nghĩa này,

$$\mathcal{H} = \bigcup_a \{h_a\} \tag{12.4}$$

có m^{r+1} phần tử.

***Định lý* 12.4**

Lớp $\mathcal{H}$ mà các phương trình (12.3) và (12.4) định nghĩa là một lớp toàn thể các hàm ánh số.

Chứng minh Xét một cặp khóa riêng biệt x, y bất kỳ. Giả sử $x_0 \neq y_0$. (Có thể tạo một đối số tương tự cho một hiệu tại bất kỳ vị trí byte nào khác.) Với bất kỳ giá trị cố định nào của $a_1, a_2, ..., a_r$, ta có chính xác một giá trị a_0 thỏa phương trình $h(x) = h(y)$; a_0 là nghiệm cho

$$a_0(x_0 - y_0) \equiv - \sum_{i=0}^{r} a_i(x_i - y_i) \pmod{m} .$$

Để xem tính chất này, ta lưu ý rằng bởi m là số nguyên tố, lượng khác zero $x_0 - y_0$ có một nghịch đảo nhân modulo m, và như vậy có một nghiệm duy nhất với a_0 modulo m. (Xem Đoạn 33.4.) Do đó, mỗi cặp

khóa x và y va chạm với chính xác m^r giá trị của a, bởi chúng va chạm chính xác một lần cho mỗi giá trị khả dĩ của $<a_1, a_2, ..., a_r>$ (tức là, với giá trị duy nhất của a_0 đã nêu trên đây). Bởi có m^{r+1} giá trị khả dĩ cho dãy a, các khóa x và y va chạm với xác suất chính xác $m^r/m^{r+1} = 1/m$. Do đó, $\mathcal{H}$ là toàn thể.

Bài tập

12.3-1

Giả sử ta muốn tìm trong một danh sách nối kết có chiều dài n, ở đó mỗi thành phần chứa một khóa k cùng với một giá trị ánh số $h(k)$. Mỗi khóa là một chuỗi ký tự dài. Làm sao để vận dụng các giá trị ánh số khi lục trong danh sách để tìm một thành phần có một khóa đã cho?

12.3-2

Giả sử một chuỗi r ký tự được ánh số vào m khe bằng cách xem nó như một số cơ số-128 rồi dùng phương pháp chia. Có thể dễ dàng biểu thị con số m dưới dạng một từ máy tính 32-bit, nhưng chuỗi r ký tự, được xem là một số cơ số-128, lấy nhiều từ. Làm sao có thể áp dụng phương pháp chia để tính toán giá trị ánh số của chuỗi ký tự mà không dùng nhiều hơn một hằng số các từ của kho lưu trữ bên ngoài chính chuỗi đó?

12.3-3

Xem xét một phiên bản của phương pháp chia qua đó $h(k) = k \bmod m$, ở đó $m = 2^p - 1$ và k là một chuỗi ký tự được diễn dịch theo cơ số 2^p. Chứng tỏ nếu có thể phái sinh chuỗi x từ chuỗi y bằng cách hoán vị các ký tự của nó, thì x và y ánh số theo cùng giá trị. Nêu một ví dụ về một ứng dụng ở đó tính chất này có thể gây rắc rối trong một hàm ánh số.

12.3-4

Xét một bảng ánh số có kích cỡ $m = 1000$ và hàm ánh số $h(k) = \lfloor m(kA \bmod 1) \rfloor$ với $A = (\sqrt{5} - 1)/2$. Tính toán các vị trí theo đó các khóa 61, 62, 63, 64, và 65 được ánh xạ.

12.3-5

Chứng tỏ nếu ta hạn chế mỗi thành phần a_i của a trong phương trình (12.3) là khác zero, thì tập hợp $\mathcal{H} = \{h_a\}$ như đã định nghĩa trong phương trình (12.4) không phải là toàn thể. (*Mách nước.* Xét các khóa $x = 0$ và $y = 1$.)

12.4 Định địa chỉ mở

Trong ***kỹ thuật định địa chỉ mở*** [open addressing], tất cả các thành phần đều được lưu trữ trong chính bảng ánh số. Nghĩa là, mỗi khoản mục trong bảng hoặc chứa một thành phần của tập hợp động hoặc chứa NIL. Khi tìm kiếm một thành phần, ta xem xét có hệ thống các khe bảng cho đến khi tìm thấy thành phần mong muốn hoặc rõ ràng thành phần đó không nằm trong bảng. Không có danh sách và thành phần nào được lưu trữ bên ngoài bảng, như trong kỹ thuật kết xích. Như vậy, trong kỹ thuật định địa chỉ mở, bảng ánh số có thể "tràn đầy" sao cho không thể tiến hành chèn thêm; hệ số tải a có thể không bao giờ vượt quá 1.

Tất nhiên, ta có thể lưu trữ các danh sách nối kết của kỹ thuật kết xích bên trong bảng ánh số, trong các khe bảng ánh số còn rảnh (xem Bài tập 12.2-5), nhưng ưu điểm của kỹ thuật định địa chỉ mở đó là tránh được các biến trỏ. Thay vì theo các biến trỏ, ta *tính toán* dãy các khe sẽ được xem xét. Bộ nhớ phụ trội được giải phóng nhờ không lưu trữ các biến trỏ sẽ cung cấp cho bảng ánh số một số lượng khe lớn hơn với cùng lượng bộ nhớ, có tiềm năng ít bị va chạm hơn và truy lục nhanh hơn.

Để thực hiện tác vụ chèn bằng kỹ thuật định địa chỉ mở, ta xem xét thành công, hoặc ***thăm dò*** [probe], bảng ánh số cho đến khi tìm ra một khe trống để đặt khóa. Thay vì được cố định theo thứ tự 0, 1,..., m -1 (yêu cầu $\Theta(n)$ thời gian tìm kiếm), dãy vị trí được thăm dò *tùy thuộc vào khóa đang được chèn.* Để xác định các khe sẽ thăm dò, ta mở rộng hàm ánh số để gộp số thăm dò (bắt đầu từ 0) dưới dạng một đầu vào thứ hai. Như vậy, hàm ánh số trở thành

$$h: U \times \{0,1,\ldots,m-1\} \rightarrow \{0,1,\ldots,m-1\}.$$

Với kỹ thuật định địa chỉ mở, ta yêu cầu rằng với mọi khóa k, ***dãy thăm dò***

$$\langle h(k, 0), h(k, 1), \ldots, h(k, m-1)\rangle$$

là một phép hoán vị của (0, 1,..., m - 1), sao cho mọi vị trí của bảng ánh số chung cuộc được xem là một khe cho một khóa mới khi bảng tràn đầy. Trong mã giả dưới đây, ta mặc nhận các thành phần trong bảng ánh số T là các khóa không có thông tin vệ tinh; khóa k đồng nhất với thành phần chứa khóa k. Mỗi khe chứa một khóa hoặc NIL (nếu khe trống).

HASH-INSERT(T, k)

```
i ← 0
repeat j ← h(k, i)
  if T[j] = NIL
      then T[j] ← k
         return j
      else i ← i + 1
  until i = m
error "hash table overflow"
```

Thuật toán để tìm khóa *k* sẽ thăm dò cùng dãy khe mà thuật toán chèn đã xem xét khi khóa *k* được chèn. Do đó, đợt tìm kiếm có thể kết thúc (không thành công) khi nó tìm một khe trống, bởi *k* ắt đã được chèn ở đó và chứ không phải về sau trong dãy thăm dò của nó. (Lưu ý, đối số này mặc nhận các khóa không được xóa ra khỏi bảng ánh số.) Thủ tục HASH-SEARCH nhận một bảng ánh số *T* và một khóa *k* làm đầu vào, trả về *j* nếu tìm thấy khe *j* chứa khóa *k*, hoặc NIL nếu khóa *k* không hiện diện trong bảng *T*.

```
HASH-SEARCH(T, k)
i ← 0
  repeat j ← h(k, i)
        if T[j] = k
           then return j
        i ← i + 1
  until T[j] = NIL hoặc i = m
return NIL
```

Xóa ra khỏi một bảng ánh số địa chỉ mở là một tiến trình khó. Khi xóa một khóa ra khỏi khe *i*, ta không thể đơn giản đánh dấu khe là trống bằng cách lưu trữ NIL trong nó. Nếu làm thế, ta không thể truy lục bất kỳ khóa *k* nào trong đợt chèn của nó mà ta đã thăm dò khe *i* và tìm thấy nó đầy. Một giải pháp đó là đánh dấu khe bằng cách lưu trữ trong nó giá trị đặc biệt DELETED thay vì NIL. Sau đó, ta phải sửa đổi thủ tục HASH-SEARCH để nó tiếp tục xem xét khi gặp giá trị DELETED, trong khi HASH-INSERT xem một khe như vậy như thể là trống sao cho có thể chèn một khóa mới. Tuy nhiên, khi làm thế, các lần tìm kiếm không còn phụ thuộc vào hệ số tải α, và vì lý do này kỹ thuật kết xích thường được lựa làm kỹ thuật giải quyết va chạm khi phải xóa các khóa.

Cuộc phân tích của chúng ta thực hiện giả thiết về ***kỹ thuật ánh số***

đều [uniform hashing]. Ta mặc nhận rằng mỗi khóa được xét có khả năng như nhau để có bất kỳ trong số $m!$ phép hoán vị của $\{0,1,..., m - 1\}$ làm dãy thăm dò của nó. Kỹ thuật ánh số đều đã tổng quát hóa khái niệm về kỹ thuật ánh số đều đơn giản được định nghĩa trên đây cho tình huống ở đó hàm ánh số không chỉ tạo ra một số đơn, mà là một dãy thăm dò đầy đủ. Tuy nhiên, kỹ thuật ánh số đều thực không dễ thực thi, và trong thực tế ta thường dùng các phép xấp xỉ (như kỹ thuật ánh số đôi, được định nghĩa dưới đây).

Ba kỹ thuật thường được dùng để tính toán các dãy thăm dò cần thiết cho kỹ thuật định địa chỉ mở: phương pháp thăm dò tuyến tính, thăm dò bậc hai [quadratic probing], và kỹ thuật ánh số đôi. Tất cả các kỹ thuật bảo đảm rằng $\langle h(k, 1), h(k, 2),..., h(k, m)\rangle$ là một phép hoán vị của $\langle 0, 1,..., m - 1\rangle$ cho mỗi khóa k. Tuy nhiên, không có kỹ thuật nào trong số này đáp ứng giả thiết của kỹ thuật ánh số đều, bởi chúng không thể phát sinh nhiều hơn m^2 dãy thăm dò khác nhau (thay vì $m!$ mà kỹ thuật ánh số đều yêu cầu). Kỹ thuật ánh số đôi có số lượng dãy thăm dò lớn nhất và, như đã dự kiến, dường như cho ta các kết quả tốt nhất.

Phương pháp thăm dò tuyến tính

Cho một hàm ánh số thường $h' : U \to \{0, 1,..., m - 1\}$, phương pháp ***thăm dò tuyến tính*** [linear probing] sử dụng hàm ánh số

$$h(k, i) = (h'(k) + i) \bmod m$$

với $i = 0,1,..., m - 1$. Cho khóa k, khe đầu tiên được thăm dò là $T[h'(k)]$. Kế tiếp ta thăm dò khe $T[h'(k) + 1]$, và tiếp tục cho lên đến khe $T[m - 1]$. Sau đó ta đóng khung [wrap around] theo các khe $T[0]$, $T[1]$,..., cho đến khi chung cuộc thăm dò khe $T[h'(k) - 1]$. Do vị trí thăm dò ban đầu xác định nguyên cả dãy thăm dò, nên chỉ có m dãy thăm dò riêng biệt được dùng với phương pháp thăm dò tuyến tính.

Phương pháp thăm dò tuyến tính dễ thực thi, song nó phải chịu một vấn đề mệnh danh là ***kết cụm sơ cấp*** [primary clustering]. Các đợt chạy kéo dài của các khe đầy sẽ tăng lên, làm tăng thời gian tìm kiếm trung bình. Ví dụ, nếu ta có $n = m/2$ khóa trong bảng, ở đó mọi khe lập chỉ mục chẵn đã đầy và mọi khe lập chỉ mục lẻ còn trống, thì đợt tìm kiếm trung bình không thành công sẽ mất 1.5 thăm dò. Tuy nhiên, nếu $n = m/2$ vị trí đầu tiên đã đầy, số lần thăm dò trung bình sẽ gia tăng lên khoảng $n/4 = m/8$. Các cụm [clusters] ắt sẽ nảy sinh, bởi nếu trước một khe

trống là i khe đầy, thì xác suất mà khe trống là khe kế tiếp của một khe đầy sẽ là $(i + 1)/m$, so sánh với một xác suất $1/m$ nếu khe đứng trước là trống. Như vậy, các đợt chạy của các khe đầy có khuynh hướng kéo dài hơn, và phương pháp thăm dò tuyến tính không phải là một phép xấp xỉ thật tốt đối với kỹ thuật ánh số đều.

Thăm dò bậc hai

Phương pháp ***thăm dò bậc hai*** sử dụng một hàm ánh số theo dạng

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m , \qquad (12.5)$$

ở đó (như trong phương pháp thăm dò tuyến tính) h' là một hàm ánh số phụ, c_1 và $c_2 \neq 0$ là các hằng phụ, và $i = 0,1,..., m -1$. Vị trí ban đầu được thăm dò là $T[h'(k)]$; các vị trí về sau là độ dịch vị theo các lượng tùy thuộc vào cách bậc hai [quadratic] trên số thăm dò i. Phương pháp này làm việc tốt hơn phương pháp thăm dò tuyến tính, song để tận dụng đầy đủ bảng ánh số, các giá trị của c_1, c_2, và m bị ràng buộc. Bài toán 12-4 có nêu một cách để lựa các tham số này. Ngoài ra, nếu hai khóa có cùng vị trí thăm dò ban đầu, thì các dãy thăm dò của chúng là như nhau, bởi $h (k_1, 0) = h (k_2, 0)$ hàm ý $h(k_1, i) = h(k_2, i)$. Điều này dẫn đến một dạng ôn hòa hơn của tính năng kết cụm, có tên ***kết cụm thứ cấp*** [secondary clustering]. Cũng như trong phương pháp thăm dò tuyến tính, đợt thăm dò ban đầu sẽ xác định nguyên cả dãy, do đó chỉ m dãy thăm dò riêng biệt được dùng.

Kỹ thuật ánh số đôi

Kỹ thuật ánh số đôi là một trong các phương pháp tốt nhất sẵn có đối với kỹ thuật định địa chỉ mở bởi vì các phép hoán vị tạo ra đều có nhiều đặc tính của các phép hoán vị được chọn ngẫu nhiên. ***Kỹ thuật ánh số đôi*** [double hashing] sử dụng một hàm ánh số có dạng

$$h (k, i) = (h_1 (k) + ih_2 (k)) \bmod m ,$$

ở đó h_1 và h_2 là các hàm ánh số phụ. Vị trí ban đầu được thăm dò là $T[h_1 (k)]$; các vị trí thăm dò kế tiếp là độ dịch vị từ các vị trí trước đó theo một lượng $h_2(k)$, modulo m. Như vậy, khác với trường hợp của thăm dò tuyến tính hoặc bậc hai, dãy thăm dò ở đây tùy thuộc vào khóa k theo hai cách, bởi vị trí thăm dò bắt đầu, độ dịch vị, hoặc cả hai, có thể thay đổi. Hình 12.5 có cung cấp một ví dụ về tác vụ chèn bằng kỹ thuật ánh số đôi.

Hình 12.5 Chèn bằng kỹ thuật ánh số đôi. Ở đây ta có một bảng ánh số có kích cỡ 13 với $h_1(k) = k \bmod 13$ và $h_2(k) = 1 + (k \bmod 11)$. Bởi 14 - 1 mod 13 và $14 \equiv 3 \bmod 11$, khóa 14 sẽ được chèn vào khe trống 9, sau khi các khe 1 và 5 đã được xét và thấy đã đầy.

Giá trị $h_2(k)$ phải là nguyên tố cùng nhau [relatively prime] đối với kích cỡ bảng ánh số m của nguyên cả bảng ánh số sẽ được tìm kiếm. Bằng không, nếu m và $h_2(k)$ có số chia chung lớn nhất $d > 1$ với một khóa k, thì một đợt tìm kiếm khóa k sẽ chỉ xem xét thứ $(1/d)$ của bảng ánh số. (Xem Chương 33.) Một cách tiện dụng để bảo đảm điều kiện này đó là cho m là một lũy thừa của 2 và thiết kế h_2 sao cho nó luôn tạo ra một số lẻ. Một cách khác đó là cho m là số nguyên tố và thiết kế h_2 sao cho nó luôn trả về một số nguyên dương nhỏ hơn m. Ví dụ, ta có thể chọn số nguyên tố m và cho

$$h_1(k) = k \bmod m ,$$

$$h_2(k) = 1 + (k \bmod m') ,$$

ở đó m' được chọn hơi nhỏ hơn m (giả sử, $m - 1$ hoặc $m - 2$). Ví dụ, nếu $k = 123456$ và $m = 701$, ta có $h_1(k) = 80$ và $h_2(k) = 257$, do đó lần thăm đầu tiên đó là định vị 80, rồi mọi khe thứ 257 (modulo m) được xem xét cho đến khi tìm thấy khóa hoặc mọi khe được xem xét.

Kỹ thuật ánh số đôi là một sự cải thiện so với thăm dò tuyến tính hoặc bậc hai ở chỗ $\Theta(m^2)$ dãy thăm dò được dùng, thay vì $\Theta(m)$, bởi từng cặp $(h_1(k), h_2(k))$ khả dĩ cho ra một dãy thăm dò riêng biệt, và khi ta thay đổi khóa, vị trí thăm dò ban đầu $h_1(k)$ và độ dịch vị $h_2(k)$ có thể thay đổi một cách độc lập. Kết quả là, khả năng thực hiện của kỹ thuật ánh số đôi tỏ ra rất sát với khả năng thực hiện của lược đồ "lý tưởng" của kỹ thuật ánh số đều.

Phân tích kỹ thuật ánh số địa chỉ mở

Giống như cuộc phân tích kết xích, cuộc phân tích kỹ thuật định địa chỉ mở cũng được diễn tả theo dạng hệ số tải α của bảng ánh số, vì n và m tiến đến vô cực. Chắc bạn còn nhớ, nếu n thành phần được lưu trữ trong một bảng có m khe, số thành phần trung bình mỗi khe là $\alpha = n/m$. Tất nhiên, với kỹ thuật định địa chỉ mở, ta có tối đa một thành phần mỗi khe, và như vậy $n \le m$, hàm ý $\alpha \le 1$.

Ta mặc nhận kỹ thuật ánh số đều được dùng. Trong lược đồ lý tưởng hóa này, dãy thăm dò $\langle h(k, 0), h(k, 1),\ldots, h(k, m - 1)\rangle$ của mỗi khóa k có khả năng như nhau là một phép hoán vị bất kỳ trên $\langle 0, 1,\ldots, m - 1\rangle$. Nghĩa là, mỗi dãy thăm dò khả dĩ có khả năng như nhau được dùng làm dãy thăm dò cho một đợt chèn hoặc tìm kiếm. Tất nhiên, một khóa đã cho có một dãy thăm dò cố định duy nhất kết hợp với nó; ý nghĩa ở đây là, xét phép phân phối xác suất trên không gian của các khóa và phép toán của hàm ánh số trên các khóa, mỗi dãy thăm dò khả dĩ có khả năng như nhau.

Giờ đây, ta phân tích số lần thăm dò dự trù của kỹ thuật ánh số bằng phương pháp định địa chỉ mở dưới giả thiết của kỹ thuật ánh số đều, ban đầu bằng một cuộc phân tích số lần thăm dò thực hiện trong một đợt tìm kiếm không thành công.

***Định lý* 12.5**

Cho một bảng ánh số địa chỉ mở với hệ số tải $\alpha = n/m < 1$, số lần thăm dò dự trù trong một đợt tìm kiếm không thành công tối đa là $1/(1 - \alpha)$, mặc nhận kỹ thuật ánh số đều.

Chứng minh Trong một đợt tìm kiếm không thành công, mọi thăm dò ngoại trừ lần thăm dò chót đều truy cập một khe đầy không chứa khóa muốn có, và khe cuối được thăm dò là trống. Ta hãy định nghĩa

$p_i = \Pr\{$chính xác i lần thăm dò truy cập các khe đầy$\}$

với $i = 0, 1, 2,\ldots$. Với $i > n$, ta có $p_i = 0$, bởi ta có thể tìm ra tối đa n khe đã đầy sẵn. Như vậy, số lần thăm dò dự trù là

$$1 + \sum_{i=0}^{\infty} ip_i\,. \qquad (12.6)$$

Để đánh giá phương trình (12.6), ta định nghĩa

$q_i = \Pr\{$ít nhất i lần thăm dò truy cập các khe đầy$\}$

với $i = 0, 1, 2,\ldots$. Như vậy, ta có thể dùng đồng nhất thức (6.28):

$$\sum_{i=0}^{\infty} ip_i = \sum_{i=0}^{\infty} q_i\,.$$

Đâu là giá trị của q_i với $i \geq 1$? Xác suất mà lần thăm dò đầu tiên truy cập một khe đầy là n / m; như vậy,

$$q_1 = \frac{n}{m} .$$

Với kỹ thuật ánh số đều, một lần thăm dò thứ hai, nếu cần, xảy ra cho một trong số m - 1 khe còn lại chưa thăm dò, n - 1 trong số đó đã đầy. Ta chỉ tạo một lần thăm dò thứ hai nếu lần thăm dò đầu tiên truy cập một khe đã đầy; như vậy,

$$q_2 = \left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right) .$$

Nói chung, lần thăm dò thứ i chỉ được thực hiện nếu i - 1 lần thăm dò đầu tiên truy cập các khe đầy, và khe được thăm dò có khả năng như nhau là bất kỳ trong số $m - i + 1$ khe còn lại, $n - i + 1$ trong số đó đã đầy. Như vậy,

$$\begin{aligned} q_i &= \left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right) \cdots \left(\frac{n-i+1}{m-i+1}\right) \\ &\leq \left(\frac{n}{m}\right)^i \\ &= \alpha^i \end{aligned}$$

với $i = 1, 2, \ldots, n$, bởi $(n - j)/(m - j) \leq n/m$ nếu $n \leq m$ và $j \geq 0$. Sau khi n lần thăm dò, tất cả n khe đầy đã được xem và sẽ không được thăm dò lại, như vậy $q_i = 0$ với $i > n$.

Giờ đây, ta sẵn sàng đánh giá phương trình (12.6). Cho giả thiết $\alpha < 1$, số lần thăm dò trung bình trong một đợt tìm kiếm không thành công là

$$\begin{aligned} 1 + \sum_{i=0}^{\infty} i p_i &= 1 + \sum_{i=1}^{\infty} q_i \\ &\leq 1 + \alpha + \alpha^2 + \alpha^3 + \ldots \\ &= \frac{1}{1 - \alpha} . \end{aligned} \qquad (12.7)$$

Phương trình (12.7) có một sự diễn dịch trực giác: một lần thăm dò luôn được thực hiện, với xác suất xấp xỉ α một lần thăm dò thứ hai là cần thiết, với xác suất xấp xỉ α^2 một lần thăm dò thứ ba là cần thiết, và vân vân.

Nếu α là một hằng, Định lý 12.5 dự báo một đợt tìm kiếm không thành công sẽ chạy trong $O(1)$ thời gian. Ví dụ, nếu bảng ánh số đầy phân nửa, số lần thăm dò trung bình trong một đợt tìm kiếm không thành công là $1 / (1 - .5) = 2$. Nếu nó là 90 phần trăm đầy, số lần thăm dò

trung bình là 1 / (1 - .9) = 10.

Định lý 12.5 cho ta khả năng thực hiện của thủ tục HASH-INSERT hầu như tức thời.

Hệ luận 12.6

Tiến trình chèn một thành phần vào một bảng ánh số địa chỉ mở có hệ số tải α yêu cầu tối đa $1/(1-\alpha)$ lần thăm dò theo trung bình, mặc nhận kỹ thuật ánh số đều.

Chứng minh Một thành phần chỉ được chèn nếu có chỗ trong bảng, và như vậy $\alpha < 1$. Tiến trình chèn một khóa yêu cầu một đợt tìm kiếm không thành công theo sau là việc sắp đặt khóa trong khe trống đầu tiên tìm thấy. Như vậy, số lần thăm dò dự trù là $1/(1-\alpha)$.

Việc tính toán số lần thăm dò dự trù cho một đợt tìm kiếm thành công yêu cầu phải tiến hành nhiều việc hơn.

Định lý 12.7

Cho một bảng ánh số địa chỉ mở có hệ số tải $\alpha < 1$, số lần thăm dò dự trù trong một đợt tìm kiếm thành công tối đa là

$$\frac{1}{\alpha}\ln\frac{1}{1-\alpha} + \frac{1}{\alpha},$$

mặc nhận kỹ thuật ánh số đều và mặc nhận mỗi khóa trong bảng có khả năng như nhau để tìm kiếm.

Chứng minh Một đợt tìm kiếm một khóa *k* theo cùng dãy thăm dò như đã được theo khi chèn thành phần có khóa *k*. Theo Hệ luận 12.6, nếu *k* là khóa thứ $(i + 1)$ được chèn vào bảng ánh số, số lần thăm dò dự trù được thực hiện trong một đợt tìm kiếm *k* tối đa là $1/(1 - i/m) = m/(m - i)$.

Nếu lấy trung bình trên tất cả *n* khóa trong bảng ánh số, ta sẽ có số lần thăm dò trung bình trong một đợt tìm kiếm thành công:

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} = \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i}$$

$$= \frac{1}{\alpha}(H_m - H_{m-n}),$$

ở đó $H_i = \sum_{j=1}^{i} 1/j$ là số điều hòa thứ *i* (như đã định nghĩa trong phương trình (3.5)). Dùng các cận trong $i \leq H_i \leq \ln i + 1$ từ các phương trình (3.11) và (3.12), ta được

$$\frac{1}{\alpha}(H_m - H_{m-n}) = \frac{1}{\alpha}(\ln m + 1 - \ln(m - n))$$
$$= \frac{1}{\alpha}\ln\frac{m}{m-n} + \frac{1}{\alpha}$$
$$= \frac{1}{\alpha}\ln\frac{1}{1-\alpha} + \frac{1}{\alpha}$$

với một cận trên số lần thăm dò dự trù trong một đợt tìm kiếm thành công.

Nếu bảng ánh số đầy phân nửa, số lần thăm dò dự trù sẽ nhỏ hơn 3.387. Nếu bảng ánh số đầy 90 phần trăm, số lần thăm dò dự trù sẽ nhỏ hơn 3.670.

Bài tập

12.4-1

Xét tiến trình chèn các khóa 10, 22, 31, 4,15, 28,17, 88, 59 vào một bảng ánh số có chiều dài $m = 11$ dùng kỹ thuật định địa chỉ mở với hàm ánh số sơ cấp $h'(k) = k \bmod m$. Minh họa kết quả của việc chèn các khóa này bằng kỹ thuật thăm dò tuyến tính, thăm dò bậc hai với $c_1 = 1$ và $c_2 = 3$, và dùng kỹ thuật ánh số đôi với $h_2(k) = 1 + (k \bmod (m - 1))$.

12.4-2

Viết mã giả cho HASH-DELETE như đã nêu khái quát trong văn bản, và sửa đổi HASH-INSERT và HASH-SEARCH để sát nhập giá trị đặc biệt DELETED.

12.4-3 *

Giả sử ta dùng kỹ thuật ánh số đôi để giải quyết các va chạm; nghĩa là, ta dùng hàm ánh số $h(k, i) = (h_1(k) + ih_2(k)) \bmod m$. Chứng tỏ dãy thăm dò $\langle h(k, 0), h(k, 1), \ldots, h(k, m-1)\rangle$ là một phép hoán vị của dãy khe $\langle 0, 1, \ldots, m-1\rangle$ nếu và chỉ nếu $h_2(k)$ là nguyên tố cùng nhau đối với m. (*Mách nước:* Xem Chương 33.)

12.4-4

Xét một bảng ánh số địa chỉ mở bằng kỹ thuật ánh số đều và một hệ số tải $\alpha = 1/2$. Đâu là số lần thăm dò dự trù trong một đợt tìm kiếm không thành công? Đâu là số lần thăm dò dự trù trong một đợt tìm kiếm thành công? Lặp lại các phép tính này với các hệ số tải 3/4 và 7/8.

12.4-5 *

Giả sử ta chèn n khóa vào một bảng ánh số có kích cỡ m dùng kỹ

thuật định địa chỉ mở và kỹ thuật ánh số đều. Cho $p(n, m)$ là xác suất mà không có va chạm nào xảy ra. Chứng tỏ $p(n, m) \leq e^{-n(n-1)/2m}$. (*Mách nước:* Xem phương trình (2.7).) Chứng tỏ khi n vượt quá $\sqrt{m}$, xác suất của việc tránh các va chạm nhanh chóng tiến đến zero.

12.4-6 *

Cận trên sêri điều hòa có thể được cải thiện theo

$$H_n = \ln n + \gamma + \frac{\epsilon}{2n}, \qquad (12.8)$$

ở đó $\gamma = 0.5772156649$...được xem là ***hằng Euler*** [Euler's constant] và ϵ thỏa $0 < \epsilon < 1$. (Xem Knuth [121] để biết nguồn gốc.) Phép xấp xỉ đã cải thiện đối với sêri điều hòa này sẽ ảnh hưởng như thế nào đối với phát biểu và chứng minh của Định lý 12.7?

12.4-7 *

Xét một bảng ánh số địa chỉ mở có một hệ số tải α. Tìm giá trị khác zero α qua đó số lần thăm dò dự trù trong một đợt tìm kiếm không thành công gấp đôi số lần thăm dò dự trù trong một đợt tìm kiếm thành công. Dùng ước lượng $(1/\alpha)\ln(1/(1-\alpha))$ với số lần thăm dò cần thiết cho một đợt tìm kiếm thành công.

Các Bài Toán

12-1 Cận thăm dò dài nhất cho kỹ thuật ánh số

Một bảng ánh số có kích cỡ m được dùng để lưu trữ n mục, với $n \leq m/2$. Kỹ thuật định địa chỉ mở được dùng để giải quyết va chạm.

a. Mặc nhận kỹ thuật ánh số đều, chứng tỏ với $i = 1, 2, ..., n$, xác suất mà lần chèn thứ i yêu cầu chính xác trên k lần thăm dò tối đa là 2^{-k}.

b. Chứng tỏ với $i = 1, 2, ..., n$, xác suất mà lần chèn thứ i yêu cầu trên $2\lg n$ lần thăm dò tối đa là $1/n^2$.

Cho biến ngẫu nhiên X_i thể hiện số lần thăm dò mà lần chèn thứ i yêu cầu. Bạn đã chứng tỏ trong phần (b) rằng $\Pr\{X_i > 2\lg n\} \leq 1/n^2$. Cho biến ngẫu nhiên $X = \max_{1\leq i\leq n} X_i$ thể hiện số lần thăm dò tối đa mà bất kỳ trong số n lần chèn đòi hỏi.

c. Chứng tỏ $\Pr\{X > 2\lg n\} \leq 1/n$.

d. Chứng tỏ chiều dài dự trù của dãy thăm dò dài nhất là $E[X] = O(\lg n)$.

12-2 Tìm trong một tập hợp tĩnh

Bạn được yêu cầu thực thi một tập hợp động n thành phần ở đó các

khóa là các con số. Tập hợp là tĩnh (không có các phép toán INSERT hoặc DELETE), và phép toán duy nhất cần thiết đó là SEARCH. Bạn được giao một thời lượng tùy ý để xử lý trước n thành phần sao cho các phép toán SEARCH chạy nhanh chóng.

a. Chứng tỏ SEARCH có thể được thực thi trong $O(\lg n)$ thời gian ca xấu nhất không dùng kho lưu trữ phụ trội vượt quá mức cần thiết để lưu trữ chính các thành phần của tập hợp.

b. Xét tiến trình thực thi tập hợp bằng kỹ thuật ánh số địa chỉ mở trên m khe, và mặc nhận kỹ thuật ánh số đều. Đâu là lượng cực tiểu của kho lưu trữ phụ trội $m - n$ cần thiết để khả năng thực hiện trung bình của một phép toán SEARCH không thành công chí ít cũng tốt như cận trong phần (a)? Đáp án của bạn phải là một biên tiệm cận trên $m - n$ theo dạng n.

12-3 Cận kích cỡ khe cho kết xích

Giả sử ta có một bảng ánh số với n khe, có các va chạm được giải quyết bằng kỹ thuật kết xích, và giả sử n khóa được chèn vào bảng. Mỗi khóa có khả năng như nhau để được ánh số theo mỗi khe. Cho M là số lượng khóa cực đại trong bất kỳ khe nào sau khi chèn tất cả các khóa. Nhiệm vụ của bạn là chứng minh một cận trên $O(\lg n/ \lg\lg n)$ trên E $[M]$, giá trị dự trù của M.

a. Chứng tỏ xác suất Q_k mà k khóa ánh số theo một khe cụ thể được cung cấp bởi

$$Q_k = \left(\frac{1}{n}\right)^k\left(1 - \frac{1}{n}\right)^{n-k}\binom{n}{k}.$$

b. Cho P_k là xác suất mà $M = k$, nghĩa là, xác suất mà khe chứa hầu hết các khóa sẽ chứa k khóa. Chứng tỏ $P_k \leq nQ_k$.

c. Dùng phép xấp xỉ Stirling, phương trình (2.11), để nêu $Q_k < e^k/k^k$.

d. Chứng tỏ ở đó tồn tại một hằng $c > 1$ sao cho $Q_{k_0} < 1/n^3$ với $k_0 = c \lg n/ \lg\lg n$. Kết luận rằng $P_{k_0} < 1/n^2$ với $k_0 = c \lg n/ \lg\lg n$.

e. Chứng tỏ rằng

$$\mathrm{E}[M] \leq \Pr\left\{M > \frac{c \lg n}{\lg\lg n}\right\}\cdot n + \Pr\left\{M \leq \frac{c \lg n}{\lg\lg n}\right\}\cdot \frac{c \lg n}{\lg\lg n}.$$

Kết luận E $[M] = O(\lg n/ \lg\lg n)$.

12-4 Thăm dò bậc hai

Giả sử ta được giao một khóa k để tìm kiếm trong một bảng ánh số có các vị trí 0, 1,..., m - 1, và giả sử ta có một hàm ánh số h ánh xạ không

gian khóa vào tập hợp $\{0, 1, ..., m - 1\}$. Lược đồ tìm kiếm như sau.

1. Tính toán giá trị $i \leftarrow h(k)$, và ấn định $j \leftarrow 0$.

2. Thăm dò tại vị trí i về khóa k mong muốn. Nếu tìm thấy nó, hoặc giả vị trí này trống, bạn kết thúc đợt tìm kiếm.

3. Ấn định $j \leftarrow (j + 1) \bmod m$ và $i \leftarrow (i + j) \bmod m$, và trả về bước 2.

Giả sử m là một lũy thừa của 2.

a. Chứng tỏ lược đồ này là một minh dụ của lược đồ "thăm dò bậc hai" chung bằng cách đưa ra các hằng thích hợp c_1 và c_2 cho phương trình (12.5).

b. Chứng minh thuật toán này xem xét mọi vị trí bảng trong ca xấu nhất.

12-5 Kỹ thuật ánh số toàn thể-k

Cho $\mathcal{H} = \{h\}$ là một lớp các hàm ánh số ở đó mỗi h ánh xạ không gian U các khóa theo $\{0, 1, ..., m - 1\}$. Ta nói $\mathcal{H}$ là ***toàn thể-k*** [*k*-universal] nếu, với mọi dãy cố định k khóa riêng biệt $<x_1, x_2, ..., x_k>$ và với bất kỳ h được chọn ngẫu nhiên từ $\mathcal{H}$, dãy $<h(x_1), h(x_2), ..., h(x_k)>$ có khả năng như nhau để là bất kỳ trong số mk dãy có chiều dài k với các thành phần được rút từ $\{0, 1, ..., m - 1\}$.

a. Chứng tỏ nếu $\mathcal{H}$ là toàn thể-2, thì nó là toàn thể.

b. Chứng tỏ lớp $\mathcal{H}$ được định nghĩa trong Đoạn 12.3.3 không phải là toàn thể-2.

c. Chứng tỏ nếu ta sửa đổi phần định nghĩa của $\mathcal{H}$ trong Đoạn 12.3.3 sao cho mỗi hàm cũng chứa một số hạng không đổi b, nghĩa là, nếu ta thay $h(x)$ bằng

$$h_{a,b}(x) = a \cdot x + b,$$

thì $\mathcal{H}$ là toàn thể-2.

Ghi chú Chương

Knuth [123] và Gonnet [90] là những tài liệu tham khảo tuyệt vời để phân tích các thuật toán ánh số. Knuth cho rằng H. P. Luhn (1953) đã phát minh các bảng ánh số, cùng với phương pháp kết xích để giải quyết các va chạm. Váo khoảng cùng thời gian đó, G. M. Amdahl đã phát sinh ý tưởng về kỹ thuật định địa chỉ mở.

13 Các Cây Tìm Nhị Phân

Các cây tìm [search trees] là những cấu trúc dữ liệu hỗ trợ nhiều phép toán tập hợp động, bao gồm SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, và DELETE. Như vậy, có thể dùng một cây tìm làm từ điển lẫn hàng đợi ưu tiên.

Các phép toán căn bản trên một cây tìm nhị phân chiếm thời gian tỷ lệ với chiều cao của cây. Với một cây nhị phân hoàn chỉnh có n nút, các phép toán như vậy chạy trong $\Theta(\lg n)$ thời gian ca xấu nhất. Tuy nhiên, nếu cây là một xích tuyến tính gồm n nút, các phép toán tương tự chiếm $\Theta(n)$ thời gian ca xấu nhất. Như sẽ nêu trong Đoạn 13.4, chiều cao của một cây tìm nhị phân được xây dựng ngẫu nhiên là $O(\lg n)$, sao cho các phép toán tập hợp động căn bản chiếm $\Theta(\lg n)$ thời gian.

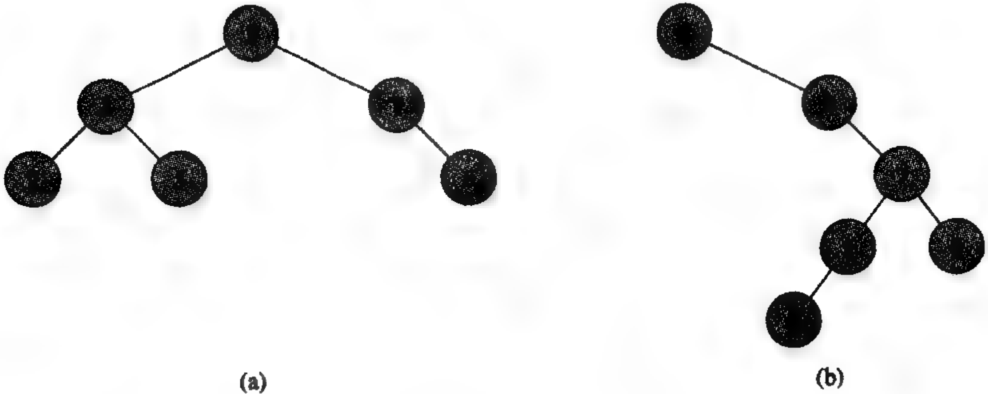
Trong thực tế, không phải lúc nào cũng có thể bảo đảm các cây tìm nhị phân được xây dựng một cách ngẫu nhiên, nhưng có các biến thể của các cây tìm nhị phân mà khả năng thực hiện ca xấu nhất của chúng trên các phép toán cơ bản có thể được bảo đảm là tốt. Chương 14 trình bày một biến thể như vậy, các cây đỏ đen, có chiều cao $O(\lg n)$. Chương 19 giới thiệu các cây B, đặc biệt tốt trong việc duy trì các cơ sở dữ liệu trên kho lưu trữ (đĩa) thứ cấp, truy cập ngẫu nhiên.

Sau khi trình bày các tính chất cơ bản của các cây tìm nhị phân, các đoạn sau sẽ nêu cách dẫn một cây tìm nhị phân để in các giá trị của nó theo thứ tự sắp xếp, cách tìm kiếm một giá trị trong một cây tìm nhị phân, cách tìm thành phần cực tiểu hoặc cực đại, cách tìm phần tử tiền vị hoặc phần tử kế vị của một thành phần, và cách chèn vào hoặc xóa ra khỏi một cây tìm nhị phân. Chương 5 đã giới thiệu các tính chất toán học căn bản của các cây.

13.1 Cây tìm nhị phân là gì?

Như tên gọi gợi ý, một cây tìm nhị phân được tổ chức theo một cây nhị phân, như đã nêu trong Hình 13.1. Một cây như vậy có thể được biểu diễn bởi một cấu trúc dữ liệu nối kết ở đó mỗi nút là một đối tượng. Ngoài một trường *key*, mỗi nút chứa các trường *left*, *right*, và *p* trỏ

đến các nút tương ứng với con trái, con phải, và cha của nó, theo thứ tự nêu trên. Nếu thiếu một con hoặc cha, trường thích hợp sẽ chứa giá trị NIL. Nút gốc là nút duy nhất trong cây có trường cha là NIL.



Hình 13.1 Các cây tìm nhị phân. Với một nút x bất kỳ, các khóa trong cây con trái của x tối đa là $key[x]$, và các khóa trong cây con phải của x ít nhất là $key[x]$. Các cây tìm nhị phân khác nhau có thể biểu diễn cùng tập hợp các giá trị. Thời gian thực hiện trường hợp xấu nhất của hầu hết các phép toán cây tìm tỷ lệ với chiều cao của cây. **(a)** Một cây tìm nhị phân trên 6 nút có chiều cao 2. **(b)** Một cây tìm nhị phân ít hiệu quả hơn có chiều cao 4 chứa cùng các khóa.

Các khóa trong một cây tìm nhị phân luôn được lưu trữ theo cách để thỏa ***tính chất cây tìm nhị phân*** [binary-search-tree property]:

Cho x là một nút trong một cây tìm nhị phân. Nếu y là một nút trong cây con trái của x, thì $key[y] \leq key[x]$. Nếu y là một nút trong cây con phải của x, thì $key[x] \leq key[y]$.

Như vậy, trong Hình 13.1(a), khóa của gốc là 5, các khóa 2, 3, và 5 trong cây con trái của nó không lớn hơn 5, và các khóa 7 và 8 trong cây con phải của nó không nhỏ hơn 5. Cùng tính chất cũng được áp dụng cho mọi nút trong cây. Ví dụ, khóa 3 trong Hình 13.1(a) không nhỏ hơn khóa 2 trong cây con trái của nó và không lớn hơn khóa 5 trong cây con phải của nó.

Tính chất cây tìm nhị phân cho phép ta in ra tất cả các khóa trong một cây tìm nhị phân theo thứ tự sắp xếp bằng một thuật toán đệ quy đơn giản, có tên là ***duyệt cây tại cấp*** [inorder tree walk]. Tên thuật toán này xuất phát từ sự việc khóa của gốc một cây con được in giữa các giá trị trong cây con trái và các giá trị trong cây con phải. (Cũng vậy, một ***duyệt cây tiền cấp*** [preorder tree walk] in gốc trước các giá trị của một trong

hai cây con, và một ***duyệt cây hậu cấp*** [postorder tree walk] in gốc sau các giá trị trong các cây con của nó.) Để dùng thủ tục dưới đây in tất cả các thành phần trong một cây tìm nhị phân T, ta gọi INORDER-TREE-WALK(*root*[T]).

```
INORDER-TREE-WALK(x)
nếu x ≠ NIL
    then INORDER-TREE-WALK(left[x])
         print key[x]
         INORDER-TREE-WALK(right[x])
```

Để lấy ví dụ, duyệt cây tại cấp in các khóa của mỗi trong số hai cây tìm nhị phân của Hình 13.1 theo thứ tự 2, 3, 5, 5, 7, 8. Tính đúng đắn của thuật toán dựa trên phương pháp quy nạp trực tiếp từ tính chất cây tìm nhị phân. Nó mất $\Theta(n)$ thời gian để rà qua một cây tìm nhị phân n-nút, bởi sau lệnh gọi ban đầu, thủ tục được gọi theo đệ quy chính xác hai lần cho mỗi nút trong cây—một lần cho con trái và một lần cho con phải.

Bài tập

13.1-1

Vẽ các cây tìm nhị phân có chiều cao 2, 3, 4, 5, và 6 trên tập hợp các khóa {1, 4, 5,10, 16,17, 21}.

13.1-2

Nêu sự khác biệt giữa tính chất cây tìm nhị phân và tính chất đống (7.1)? Có thể dùng tính chất đống để in ra các khóa của một cây n-nút theo thứ tự sắp xếp trong $O(n)$ thời gian? Giải thích cách thức hoặc tại sao không.

13.1-3

Nêu một thuật toán phi đệ quy thực hiện một duyệt cây tại cấp. (*Mách nước:* Có một giải pháp dễ dàng sử dụng một ngăn xếp làm cấu trúc dữ liệu phụ trợ và một giải pháp tuy lịch lãm song phức hợp hơn không sử dụng ngăn xếp nhưng mặc nhận có thể trắc nghiệm hai biến trỏ cho đẳng thức.)

13.1-4

Nêu các thuật toán đệ quy thực hiện các duyệt cây tiền và hậu cấp trong $\Theta(n)$ thời gian trên một cây n nút.

13.1-5

Chứng tỏ rằng do việc sắp xếp n thành phần chiếm $\Omega(n \lg n)$ thời

gian trong trường hợp xấu nhất trong mô hình so sánh, nên thuật toán gốc so sánh để kiến tạo một cây tìm nhị phân từ một danh sách tùy ý n thành phần sẽ choán $\Omega(n \lg n)$ thời gian trong trường hợp xấu nhất.

13.2 Truy vấn một cây tìm nhị phân

Với một cây tìm nhị phân, phép toán phổ dụng nhất đó là tìm kiếm một khóa được lưu trữ trong cây. Ngoài phép toán SEARCH, các cây tìm nhị phân có thể hỗ trợ các kiểu truy vấn như MINIMUM, MAXIMUM, SUCCESSOR, và PREDECESSOR. Trong đoạn này, ta xét các phép toán này và chứng tỏ từng phép toán có thể được hỗ trợ trong thời gian $O(h)$ trên một cây tìm nhị phân có chiều cao h.

Hình 13.2 Các kiểu truy vấn trên một cây tìm nhị phân. Để tìm kiếm khóa 13 trong cây, ta theo lộ trình 15 → 6 → 7 → 13 từ gốc. Khóa cực tiểu trong cây là 2, có thể tìm thấy bằng cách theo các biến trỏ *trái* từ gốc. Khóa cực đại 20 được tìm thấy bằng cách theo các biến trỏ *phải* từ gốc. Phần tử kế vị của nút có khóa 15 là nút có khóa 17, bởi nó là khóa cực tiểu trong cây con phải của 15. Nút có khóa 13 không có cây con phải, và như vậy phần tử kế vị của nó là tiền bối thấp nhất của nó có con trái cũng là một tiền bối. Trong trường hợp này, nút có khóa 15 là phần tử kế vị của nó.

Tìm kiếm

Ta dùng thủ tục dưới đây để tìm một nút có một khóa đã cho trong một cây tìm nhị phân. Cho một biến trỏ đến gốc của cây và một khóa k, TREE-SEARCH trả về một biến trỏ đến một nút có khóa k nếu như nó tồn tại; bằng không, nó trả về NIL.

TREE-SEARCH(x, k)

if x = NIL or k = *key*[x]

```
        then return x
    if k < key[x]
        then return TREE-SEARCH(left[x], k)
        else return TREE-SEARCH(right[x], k)
```

Thủ tục bắt đầu đợt tìm kiếm của nó tại gốc và lần theo lộ trình đi xuống cây, như đã nêu trong Hình 13.2. Với mỗi nút *x* đụng gặp, nó so sánh khóa *k* với *key*[*x*]. Nếu hai khóa bằng nhau, đợt tìm kiếm kết thúc. Nếu *k* nhỏ hơn *key*[*x*], đợt tìm kiếm tiếp tục trong cây con trái của *x*, bởi tính chất cây tìm nhị phân hàm ý *k* có thể không được lưu trữ trong cây con phải. Theo đối xứng, nếu *k* lớn hơn *key*[*k*], đợt tìm kiếm tiếp tục trong cây con phải. Các nút đã gặp trong đệ quy sẽ hình thành một lộ trình đi xuống dưới từ gốc của cây, và như vậy thời gian thực hiện của TREE-SEARCH là $O(h)$, ở đó *h* là chiều cao của cây.

Cùng thủ tục có thể được viết lặp lại bằng cách "bung" phép đệ quy vào một vòng lặp **while.** Trên hầu hết các máy tính, phiên bản này hiệu quả hơn.

```
ITERATIVE-TREE-SEARCH(x, k)
    while x ≠ NIL và k ≠ key[x]
        do if k < key[x]
              then x ← left[x]
           else x ← right[x]
    return x
```

Cực tiểu và cực đại

Lúc nào cũng có thể tìm thấy một thành phần trong một cây tìm nhị phân có khóa là một cực tiểu bằng cách theo các biến trỏ con *trái* từ gốc cho đến khi gặp một NIL, như đã nêu trong Hình 13.2. Thủ tục dưới đây trả về một biến trỏ đến thành phần cực tiểu trong cây con có gốc tại một nút *x* đã cho.

```
TREE-MINIMUM(x)
    while left[x] ≠ NIL
        do x ← left[x]
    return x
```

Tính chất cây tìm nhị phân bảo đảm TREE-MINIMUM là đúng đắn. Nếu một nút *x* không có cây con trái, thì do mọi khóa trong cây con phải của *x* ít nhất cũng lớn bằng *key*[*x*], nên khóa cực tiểu trong cây con có

gốc tại x sẽ là *key*[x]. Nếu nút x có một cây con trái, thì do không có khóa nào trong cây con phải nhỏ hơn *key*[x] và mọi khóa trong cây con trái không lớn hơn *key*[x], nên khóa cực tiểu trong cây con có gốc tại x có thể tìm thấy trong cây con có gốc tại *left*[x].

Mã giả của TREE-MAXIMUM là đối xứng.

```
TREE-MAXIMUM(x)
while right[x] ≠ NIL
      do x ← right[x]
return x
```

Cả hai thủ tục này chạy trong $O(h)$ thời gian trên một cây có chiều cao h, bởi chúng lần theo các lộ trình đi xuống cây.

Phần tử kế vị và phần tử tiền vị

Cho một nút trong một cây tìm nhị phân, đôi lúc ta cần tìm ra phần tử kế vị của nó theo thứ tự sắp xếp mà một duyệt cây tại cấp xác định. Nếu tất cả các khóa đều riêng biệt, phần tử kế vị của một nút x là nút có khóa nhỏ nhất lớn hơn *key*[x]. Cấu trúc của một cây tìm nhị phân cho phép ta xác định phần tử kế vị của một nút mà không hề so sánh các khóa. Thủ tục dưới đây trả về phần tử kế vị của một nút x trong một cây tìm nhị phân nếu nó tồn tại, và NIL, nếu x có khóa lớn nhất trong cây.

```
TREE-SUCCESSOR(x)
1 if right[x] ≠ NIL
2       then return TREE-MINIMUM(right[x])
3 y ← p[x]
4 while y ≠ NIL và x = right[y]
5       do x ← y
6          y ← p[y]
7 return y
```

Mã của TREE-SUCCESSOR được tách nhỏ thành hai trường hợp. Nếu cây con phải của nút x không trống, thì phần tử kế vị của x chính là nút mút trái trong cây con phải, được tìm thấy trong dòng 2 bằng cách gọi TREE-MINIMUM(*right*[x]). Ví dụ, phần tử kế vị của nút có khóa 15 trong Hình 13.2 chính là nút có khóa 17.

Mặt khác, nếu cây con phải của nút x là trống và x có một phần tử kế vị y, thì y là tiền bối thấp nhất của x có con trái cũng là một tiền bối của x. Trong Hình 13.2, phần tử kế vị của nút có khóa 13 là nút có khóa 15.

Để tìm ra y, ta đơn giản lần lên cây từ x cho đến khi gặp một nút là con trái của cha nó; điều này được thực hiện bởi các dòng 3-7 của TREE-SUCCESSOR.

Thời gian thực hiện của TREE-SUCCESSOR trên một cây có chiều cao h là $O(h)$, bởi ta theo một lộ trình đi lên cây hoặc theo một lộ trình đi xuống cây. Thủ tục TREE-PREDECESSOR, đối xứng đối với TREE-SUCCESSOR, cũng chạy trong thời gian $O(h)$.

Tóm lại, ta đã chứng minh định lý sau đây.

Định lý 13.1

Có thể thực hiện các phép toán tập hợp động SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, và PREDECESSOR để chạy trong $O(h)$ thời gian trên một cây tìm nhị phân có chiều cao h.

Bài Tập

13.2-1

Giả sử ta có các con số giữa 1 và 1000 trong một cây tìm nhị phân và muốn tìm kiếm con số 363. Dãy nào dưới đây *không thể* là dãy các nút được xem xét?

a. 2, 252, 401, 398, 330, 344, 397, 363.

b. 924, 220, 911, 244, 898, 258, 362, 363.

c. 925, 202, 911, 240, 912, 245, 363.

d. 2, 399, 387, 219, 266, 382, 381, 278, 363.

e. 935, 278, 347, 621, 299, 392, 358, 363.

13.2-2

Giáo sư Bunyan cho rằng ông đã phát hiện một tính chất đáng lưu ý của các cây tìm nhị phân. Giả sử đợt tìm kiếm khóa k trong một cây tìm nhị phân kết thúc trong một lá. Xét ba tập hợp: A, các khóa về bên trái của lộ trình tìm kiếm; B, các khóa trên lộ trình tìm kiếm; và C, các khóa về bên phải của lộ trình tìm kiếm. Giáo sư Bunyan yêu cầu bất kỳ trong số ba khóa $a \in A$, $b \in B$, và $c \in C$ phải thỏa $a \leq b \leq c$. Nêu một ví dụ ngược lại khả dĩ nhỏ nhất đối với yêu cầu của giáo sư.

13.2-3

Dùng tính chất cây tìm nhị phân để chứng minh chính xác rằng mã của TREE-SUCCESSOR là đúng.

13.2-4

Có thể thực thi một duyệt cây tại cấp của một cây tìm nhị phân n-nút

bằng cách tìm thành phần cực tiểu trong cây bằng TREE-MINIMUM rồi thực hiện n - 1 lần gọi đến TREE-SUCCESSOR. Chứng minh thuật toán này chạy trong $\Theta(n)$ thời gian.

13.2-5

Chứng minh bất kể ta bắt đầu tại nút nào trong một cây tìm nhị phân có chiều cao h, k lần gọi kế tiếp đến TREE-SUCCESSOR sẽ mất $O(k + h)$ thời gian.

13.2-6

Cho T là một cây tìm nhị phân, cho x là một nút lá, và cho y là cha của nó. Chứng tỏ *key*[*y*] là khóa nhỏ nhất trong T lớn hơn *key*[*x*] hoặc là khóa lớn nhất trong cây nhỏ hơn *key*[*x*].

13.3 Chèn và xóa

Các phép toán chèn và xóa sẽ làm thay đổi tập hợp động được biểu thị bởi một cây tìm nhị phân. Cấu trúc dữ liệu phải được sửa đổi để phản ánh thay đổi này, nhưng theo một cách mà tính chất cây tìm nhị phân vẫn có thể tiếp tục. Như sẽ thấy, việc sửa đổi cây để chèn một thành phần mới tương đối dễ hiểu, nhưng tiến trình điều quản tác vụ xóa có hơi phức tạp hơn.

Chèn

Để chèn một giá trị mới v vào một cây tìm nhị phân T, ta dùng thủ tục TREE-INSERT. Thủ tục được chuyền một nút z qua đó *key*[*z*] = v, *left*[*z*] = NIL, và *right*[*z*] = NIL. Nó sửa đổi T và vài trường của z theo cách thức mà z được chèn vào một vị trí thích hợp trong cây.

```
TREE-INSERT(T, z)
y ← NIL
x ← root[T]
while x ≠ NIL
      do y ← x
           if key[z] < key[x]
                then x ← left[x)
                else x ← right[x]
p[z] ← y
if y = NIL
```

```
10          then root[T] ← z
11          else if  key[z] < key[y]
12                   then left[y] ← z
13                   else right[y] ← z
```

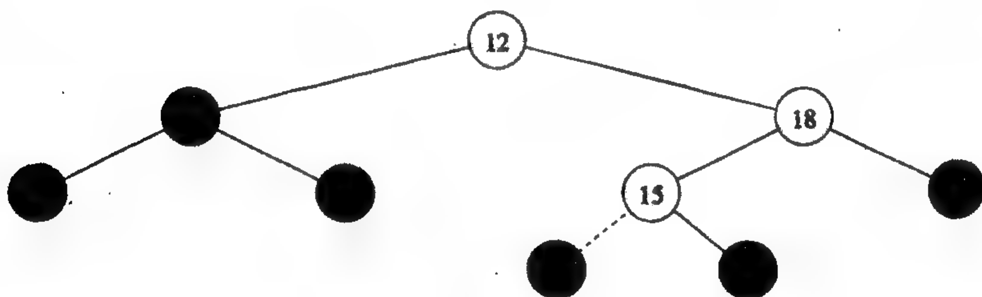
Hình 13.3 nêu cách làm việc của TREE-INSERT. Giống như các thủ tục TREE-SEARCH và ITERATIVE-TREE-SEARCH, TREE-INSERT bắt đầu tại gốc của cây và lần theo một lộ trình đi xuống. Biến trỏ *x* lần theo lộ trình, và biến trỏ *y* được duy trì dưới dạng cha của *x*. Sau khi khởi tạo, vòng lặp **while** trong các dòng 3-7 sẽ khiến hai biến trỏ này dời xuống cây, đi theo trái hay phải tùy thuộc vào phép so sánh của *key*[*z*] với *key*[*x*], cho đến khi *x* được ấn định là NIL. NIL này choán vị trí ở đó ta muốn đặt mục đầu vào *z*. Các dòng 8-13 ấn định các biến trỏ khiến *z* được chèn.

Giống như các phép toán nguyên hàm [primitive operations] khác trên các cây tìm kiếm, thủ tục TREE-INSERT chạy trong $O(h)$ thời gian trên một cây có chiều cao h.

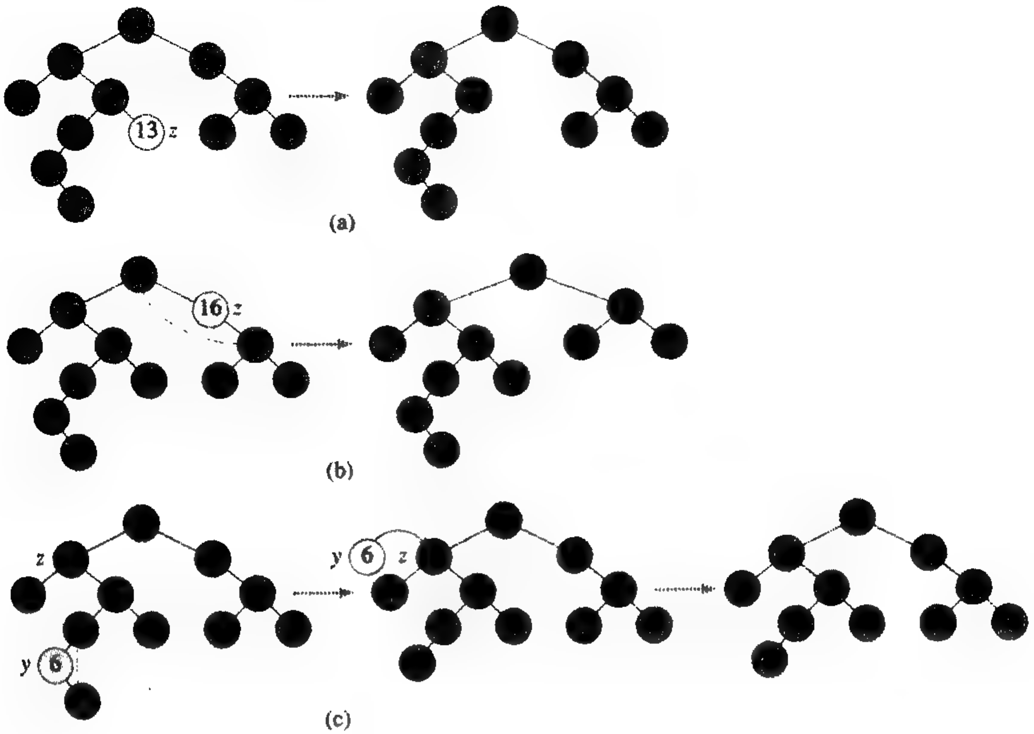
Xóa

Thủ tục để xóa một nút *z* đã cho ra khỏi một cây tìm nhị phân chấp nhận một biến trỏ đến *z* dưới dạng một đối số. Thủ tục xét ba trường hợp nêu trong Hình 13.4. Nếu *z* không có các con, ta sửa đổi cha của nó là *p*[*z*] để thay *z* bằng NIL làm con của nó. Nếu nút chỉ có một con đơn lẻ, ta "nối khử" [splice out] *z* bằng cách tạo một nối kết mới giữa con của nó và cha của nó. Cuối cùng, nếu nút có hai con, ta nối khử phần tử kế vị *y* của *z*, không có con trái (xem Bài tập 13.3-4) và thay nội dung của *z* bằng nội dung của *y*.

Mã của TREE-DELETE tổ chức ba trường hợp này có hơi khác nhau.



Hình 13.3 Chèn một mục có khóa 13 vào một cây tìm nhị phân. Các nút tô bóng sáng nêu rõ lộ trình từ gốc đổ xuống đến vị trí nơi chèn mục đó. Vạch chấm cách nêu rõ nối kết trong cây được bổ sung để chèn mục đó.



Hình 13.4 Xóa một nút z ra khỏi một cây tìm nhị phân. Trong mỗi trường hợp, nút thực tế được gỡ bỏ sẽ được tô bóng sáng. **(a)** Nếu z không có con, ta chỉ việc gỡ bỏ nó. **(b)** Nếu z chỉ có một con, ta nối khử z. **(c)** Nếu z có hai con, ta nối khử phần tử kế vị y của nó, có tối đa một con, rồi thay nội dung của z bằng nội dung của y.

```
TREE-DELETE(T, z)
if left[z] = NIL  or  right[z] = NIL
    then y ← z
    else y ← TREE-SUCCESSOR(z)
if left[y] ≠ NIL
    then x ← left[y]
    else x ← right[y]
if x ≠ NIL
    then p[x] ← p[y]
if p[y] = NIL
    then root[T] ← x
    else y = left[p[y]]
```

```
12              then left[p[y]] ← x
13              else right[p[y]] ← x
14  if y ≠ z
15      then key[z] ← key[y]
16              ▷ if y có các trường hợp khác, chép chúng.
17  return y
```

Trong các dòng 1-3, thuật toán xác định một nút y để khử. Nút y sẽ là nút đầu vào z (nếu z có tối đa 1 con) hoặc phần tử kế vị của z (nếu z có hai con). Thì, trong các dòng 4-6, x được ấn định theo con phi-NIL của y, hoặc theo NIL nếu y không có con. Nút y được nối khử trong các dòng 7-13 bằng cách sửa đổi các biến trỏ trong $p[y]$ và x. Tiến trình nối khử y có hơi phức tạp bởi nhu cầu phải điều quản đúng đắn các điều kiện cận, xảy ra khi x = NIL hoặc khi y là gốc. Cuối cùng, trong các dòng 14-16, nếu phần tử kế vị của z là nút được nối khử, nội dung của z được dời từ y sang z, ghi đè nội dung trước đó. Nút y được trả về trong dòng 17 sao cho thủ tục gọi có thể tái sinh nó thông qua danh sách rảnh. Thủ tục chạy trong $O(h)$ thời gian trên một cây có chiều cao h.

Tóm lại, ta đã chứng minh định lý sau.

Định lý 13.2

Có thể thực hiện các phép toán tập hợp động INSERT và DELETE để chạy trong $O(h)$ thời gian trên một cây tìm nhị phân có chiều cao h.

Bài tập

13.3-1

Nêu một phiên bản đệ quy của thủ tục TREE-INSERT.

13.3-2

Giả sử một cây tìm nhị phân được kiến tạo bằng cách liên tục chèn các giá trị riêng biệt vào cây. Chứng tỏ số lượng nút được xem xét trong khi tìm kiếm một giá trị trong cây là một cộng với số lượng nút được xem xét khi giá trị được chèn đầu tiên vào cây.

13.3-3

Ta có thể sắp xếp một tập hợp đã cho n con số bằng cách trước tiên xây dựng một cây tìm nhị phân chứa các con số này (dùng lặp TREE-INSERT để lần lượt chèn từng con số một) sau đó in các con số bằng một duyệt cây tại cấp. Đâu là thời gian thực hiện trường hợp xấu nhất và trường hợp tốt nhất cho thuật toán sắp xếp này?

13.3-4

Chứng tỏ nếu một nút trong một cây tìm nhị phân có hai con, thì phần tử kế vị của nó không có con trái và phần tử tiền vị của nó không có con phải.

13.3-5

Giả sử một cấu trúc dữ liệu khác chứa một biến trỏ đến một nút *y* trong một cây tìm nhị phân, và giả sử phần tử tiền vị *z* của *y* được xóa ra khỏi cây bởi thủ tục TREE-DELETE. Có thể nảy sinh vấn đề gì? Có thể viết lại TREE-DELETE như thế nào để giải quyết bài toán này?

13.3-6

Phép toán xóa có "giao hoán" hay không theo nghĩa tiến trình xóa *x* rồi *y* ra khỏi một cây tìm nhị phân sẽ rời cùng cây như khi xóa *y* rồi *x*? Lập luận tại sao nó là hoặc cho một vấn đề ngược lại.

13.3-7

Khi nút *z* trong TREE-DELETE có hai con, ta có thể nối khử phần tử tiền vị của nó thay vì phần tử kế vị. Có vài người đã lập luận rằng một chiến lược công bằng, gán mức ưu tiên bằng nhau cho phần tử tiền vị lẫn phần tử kế vị, sẽ cho ra khả năng thực hiện theo kinh nghiệm tốt hơn. Có thể thay đổi TREE-DELETE như thế nào để thực thi một chiến lược công bằng như vậy?

---

* 13.4 Các cây tìm nhị phân được xây dựng ngẫu nhiên

Ta đã chứng tỏ tất cả các phép toán cơ bản trên một cây tìm nhị phân chạy trong $O(h)$ thời gian, ở đó h là chiều cao của cây. Tuy nhiên, chiều cao của một cây tìm nhị phân thay đổi, khi các mục được chèn và được xóa. Để phân tích cách ứng xử của các cây tìm nhị phân trong thực tế, ta có thể thực hiện giả thiết thống kê về phép phân phối các khóa và dãy các phép chèn và xóa.

Đáng tiếc, ta không biết nhiều về chiều cao trung bình của một cây tìm nhị phân khi dùng cả phép chèn lẫn xóa để tạo nó. Khi cây được tạo bởi một mình phép chèn, tiến trình phân tích thường dễ theo dõi hơn. Do đó, ta hãy định nghĩa một ***cây tìm nhị phân xây dựng theo ngẫu nhiên*** trên n khóa riêng biệt như trường hợp chèn các khóa theo thứ tự ngẫu nhiên vào một cây trống từ đầu, ở đó mỗi trong số $n!$ phép hoán vị của các khóa đầu vào có khả năng như nhau. (Bài tập 13.4-2 yêu cầu bạn chứng tỏ khái niệm này khác với việc giả định mọi cây tìm nhị phân trên n khóa có khả năng như nhau.) Mục tiêu của đoạn này đó là chứng

tỏ chiều cao dự trù của một cây tìm nhị phân xây dựng ngẫu nhiên trên n khóa là $O(\lg n)$.

Ta bắt đầu bằng cách điều tra cấu trúc của các cây tìm nhị phân được xây dựng bằng phép chèn mà thôi.

Bổ đề 13.3

Cho T là cây xuất xứ từ việc chèn n khóa riêng biệt $k_1, k_2, \ldots, k_n$ (theo thứ tự) vào một cây tìm nhị phân trống từ đầu. Thì k_i là một tiền bối của k_j trong T, với $1 \leq i \leq j \leq n$, nếu và chỉ nếu

$$k_i = \min \{k_l : 1 \leq l \leq i \text{ và } k_l > k_j\}$$

hoặc

$$k_i = \max \{k_l : 1 \leq l \leq i \text{ và } k_l < k_j\}.$$

Chứng minh $\Rightarrow$: Giả sử k_i là một tiền bối của k_j. Xét cây T_i là kết quả sau khi chèn các khóa $k_1, k_2, \ldots, k_i$. Lộ trình trong T_i từ gốc các k_i giống như lộ trình trong T từ gốc đến k_i. Như vậy, nếu k_j đã được chèn vào T_i, nó sẽ trở thành con trái hoặc con phải của k_i. Do vậy (xem Bài tập 13.2-6), k_i là khóa nhỏ nhất trong số $k_1, k_2, \ldots, k_i$ lớn hơn k_j hoặc là khóa lớn nhất trong số $k_1, k_2, \ldots, k_i$ nhỏ hơn k_j.

$\Leftarrow$ Giả sử k_i là khóa nhỏ nhất trong số $k_1, k_2, \ldots, k_i$ lớn hơn k_j. (Trường hợp ở đó k_i là khóa lớn nhất trong số $k_1, k_2, \ldots, k_i$ nhỏ hơn k_j được điều quản theo đối xứng.) Tiến trình so sánh k_j với bất kỳ khóa nào trên lộ trình trong T từ gốc đến k_i sẽ cho ra cùng kết quả như khi so sánh k_i với các khóa. Do đó, khi k_j được chèn, nó theo một lộ trình qua k_i và được chèn dưới dạng một hậu duệ của k_i.

Như là một hệ luận của Bổ đề 13.3, ta có thể đặc trưng hóa chính xác chiều sâu của một khóa dựa trên phép hoán vị đầu vào.

Hệ luận 13.4

Cho T là cây xuất xứ từ tiến trình chèn n khóa riêng biệt $k_1, k_2, \ldots, k_n$ (theo thứ tự) vào một cây tìm nhị phân trống từ đầu. Với một khóa đã cho k_j, ở đó $1 \leq j \leq n$, hãy định nghĩa

$$G_j = \{k_i : 1 \leq i < j \text{ và } k_l > k_i > k_j \text{ với tất cả } l < i \text{ sao cho } k_l > k_j\}$$

và

$$L_j = \{k_i : 1 \leq i < j \text{ và } k_l < k_i < k_j \text{ với tất cả } l < i \text{ sao cho } k_l < k_j\}.$$

Thì các khóa trên lộ trình từ gốc đến k_j chính xác là những khóa trong $G_j \cup L_j$, và chiều sâu trong T của bất kỳ khóa k_j nào là

$$d(k_j, T) = |G_j| + |L_j|.$$

Hình 13.5 minh họa hai tập hợp G_j và L_j. Tập hợp G_j chứa bất kỳ khóa k_i nào được chèn trước k_j sao cho k_i là khóa nhỏ nhất trong số k_1, k_2,..., k_i lớn hơn k_j. (Cấu trúc của L_j là đối xứng.) Để hiểu rõ hơn về tập hợp G_j, ta hãy khảo sát một phương pháp qua đó có thể điểm danh [enumerate] các thành phần của nó. Trong số các khóa k_1, k_2,..., k_{j-1}, hãy xét theo thứ tự các khóa lớn hơn k_j. Các khóa này được nêu dưới dạng G_j trong hình. Khi lần lượt xem xét từng khóa, bạn liên tục ghi sổ cực tiểu. Tập hợp G_j bao gồm các thành phần cập nhật cực tiểu liên tục [running minimum].

Ta hãy rút gọn bớt bối cảnh này cho mục tiêu phân tích. Giả sử n con số riêng biệt được lần lượt chèn vào một tập hợp động. Nếu tất cả các phép hoán vị của các con số có khả năng như nhau, cực tiểu của tập hợp sẽ thay đổi bao nhiêu lần, tính theo trung bình? Để trả lời câu hỏi này, giả sử số thứ i được chèn là k_i, với $i = 1, 2,..., n$. Xác suất là $1/i$ cho thấy k_i là cực tiểu của i con số đầu tiên, bởi hạng của k_i trong số i con số đầu tiên có khả năng như nhau để là bất kỳ trong số i hạng khả dĩ. Bởi vậy, số lần thay đổi dự trù đối với cực tiểu của tập hợp là

$$\sum_{i=1}^{n} \frac{1}{n} = H_n ,$$

ở đó $H_n = \ln n + O(1)$ là số điều hòa thứ n (xem phương trình (3.5) và Bài toán 6-2).

Do đó ta mong đợi số lần thay đổi đối với cực tiểu sẽ xấp xỉ là $\ln n$, và bổ đề dưới đây chứng tỏ xác suất cho thấy nó lớn hơn nhiều lại rất nhỏ.

***Bổ đề* 13.5**

Cho k_1, k_2,..., k_n là một phép hoán vị ngẫu nhiên của n con số riêng biệt, và cho $|S|$ là biến ngẫu nhiên là bản số của tập hợp

$$S = \{k_i : 1 \leq i \leq n \text{ and } k_l > k_i \text{ for all } 1 < i\}. \qquad (13.1)$$

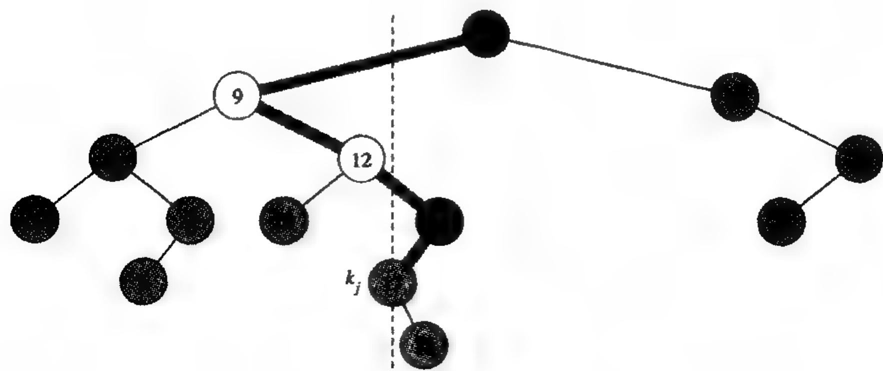
Thì $\Pr\{|S| \geq (\beta + 1)H_n\} \leq 1/n^2$, ở đó H_n là số điều hòa thứ n và $\beta \approx 4.32$ thỏa phương trình $(\ln \beta - 1)\beta = 2$.

Chứng minh Ta có thể xem bản số của tập hợp S là đang được xác định bởi n lần thử Bernoulli, ở đó một lần thành công xảy ra trong lần thử thứ i khi k_i nhỏ hơn các thành phần k_1, k_2,..., k_{i-1} . Lần thành công trong lần thử thứ i xảy ra với xác suất $1/i$. Các lần thử là độc lập, bởi xác suất cho thấy k_i là cực tiểu của k_j, k_2,..., k_i độc lập với cách sắp xếp thứ tự tương đối của k_1, k_2,...,k_{i-1}.

Ta có thể dùng Định lý 6.6 để định cận xác suất cho thấy $|S| \geq (\beta + 1)H_n$. Giá trị kỳ vọng của $|S|$ là $\mu = H_n \geq \ln n$. Bởi $\beta > 1$, nên Định lý 6.6 cho ra

$$
\begin{aligned}
\Pr\{|S| \geq (\beta+1)H_n\} &= \Pr\{|S| - \mu \geq \beta H_n\} \\
&\leq \left(\frac{eH_n}{\beta H_n}\right)^{\beta H_n} \\
&= e^{(1-\ln\beta)\beta H_n} \\
&\leq e^{-(\ln\beta-1)\beta\ln n} \\
&= n^{-(\ln\beta-1)\beta} \\
&= 1/n^2 ,
\end{aligned}
$$

là do phần định nghĩa của β .



(a)

Key	21	9	4	25	7	12	3	10	19	29	17	6	26	18
G'_j	21		25						19	29				
G_j	21								19					
L'_j		9			7	12	3	10						
L_j		9				12								

(b)

Hình 13.5 Minh họa hai tập hợp G_j và L_j bao gồm các khóa trên một lộ trình từ gốc của một cây tìm nhị phân đến một khóa $k_j = 17$. **(a)** Các nút có các khóa trong G_j được tô đen, và các nút có các khóa trong L_j mang màu trắng. Tất cả các nút khác được tô bóng. Lộ trình từ gốc đổ xuống đến nút có khóa k_j được tô bóng. Các khóa về bên trái của vạch chấm cách nhỏ hơn k_j, và các khóa về bên phải lớn hơn. Cây được kiến tạo bằng cách chèn các khóa như đã nêu trong danh sách trên cùng trong **(b)**. Tập hợp $G'_j = \{21, 25, 19, 29\}$ bao gồm các thành phần được chèn trước 17 và lớn hơn 17. Tập hợp $G_j = \{21, 19\}$ bao gồm các thành phần cập nhật một cực tiểu liên tục của các thành phần trong G'_j. Như vậy, khóa 21 nằm trong G_j, bởi nó là thành phần đầu tiên. Khóa 25 không trong G_j, bởi nó lớn hơn cực tiểu liên tục 21. Khóa 19 nằm trong G_j, bởi nó nhỏ hơn cực tiểu liên tục 21. Khóa 29 không nằm trong G_j, bởi nó lớn hơn cực tiểu liên tục 19. Các cấu trúc của L'_j và L_j là đối xứng.

Giờ đây ta có các công cụ để định cận chiều cao của một cây tìm nhị phân xây dựng ngẫu nhiên.

Định lý 13.6

Chiều cao trung bình của một cây tìm nhị phân xây dựng ngẫu nhiên trên n khóa riêng biệt là $O(\lg n)$.

Chứng minh Cho $k_1, k_2, \ldots, k_n$ là một phép hoán vị ngẫu nhiên trên n khóa, và cho T là cây tìm nhị phân xuất xứ từ việc chèn các khóa theo thứ tự vào một cây trống từ đầu. Trước tiên ta xét xác suất mà chiều sâu $d(k_j, T)$ của một khóa đã cho k_j ít nhất là t, với một giá trị tùy ý t. Qua việc mô tả đặc tính của $d(k_j, T)$ trong Hệ luận 13.4, nếu chiều sâu của k_j ít nhất là t, thì bản số của một trong hai tập hợp G_j và L_j phải ít nhất là $t/2$. Như vậy,

$$\Pr\{d(k_j, T) \geq t\} \leq \Pr\{|G_j| \geq t/2\} + \Pr\{|L_j| \geq t/2\} . \qquad (13.2)$$

Hãy xem xét $\Pr\{|G_j| \geq t/2\}$ trước. Ta có

$$\begin{aligned}\Pr\{|G_j| \geq t/2\} &= \Pr\{|\{k_i : 1 \leq i < j \text{ and } k_l > k_i > k_j \text{ for all } l < i\}| \geq t/2\} \\ &= \Pr\{|\{k_i : i \leq n \text{ and } k_l > k_i \text{ for all } l < i\}| \geq t/2\} \\ &= \Pr\{|S| \geq t/2\} ,\end{aligned}$$

ở đó S được định nghĩa như trong phương trình (13.1). Để xác minh đối số này, hãy lưu ý xác suất không giảm nếu ta mở rộng miền giá trị của i từ $i < j$ đến $i \leq n$, do có thêm các thành phần được bổ sung vào tập hợp. Cũng vậy, xác suất không giảm nếu ta gỡ bỏ điều kiện rằng $k_i > k_j$, do ta đang dùng một phép hoán vị ngẫu nhiên trên khả dĩ ít hơn n thành phần (các k_i lớn hơn k_j) để thay cho một phép hoán vị ngẫu nhiên trên n thành phần.

Dùng một đối số đối xứng, ta có thể chứng minh rằng

$$\Pr\{|L_j| \geq t/2\} \leq \Pr\{|S| \geq t/2\} ,$$

và như vậy, qua bất đẳng thức (13.2), ta được

$$\Pr\{d(k_j, T) \geq t\} \leq 2\Pr\{|S| \geq t/2\} .$$

Nếu chọn $t = 2(\beta + 1)H_n$, ở đó H_n là số điều hòa thứ n và ($\beta \approx 4.32$) thỏa $(\ln \beta - 1)\beta = 2$, ta có thể áp dụng Bổ đề 13.5 để kết luận rằng

$$\begin{aligned}\Pr\{d(k_j, T) \geq 2(\beta + 1)H_n\} &\leq 2\Pr\{|S| (\beta + 1)H_n\} \\ &\leq 2/n^2 .\end{aligned}$$

Bởi có tối đa n nút trong một cây tìm nhị phân xây dựng ngẫu nhiên, nên xác suất cho thấy chiều sâu của một nút *bất kỳ* ít nhất là $2(\beta + 1)H_n$, với bất đẳng thức Boole (6.22), tối đa là $n(2/n^2) = 2/n$. Như vậy, ít nhất 1

- $2/n$ của thời gian, chiều cao của một cây tìm nhị phân xây dựng ngẫu nhiên sẽ nhỏ hơn $2(\beta + 1)H_n$ và tối đa là $2/n$ của thời gian, nó tối đa là n. Do đó chiều cao dự trù tối đa là $(2(\beta + 1)H_n)(1 - 2/n) + n(2/n) = O(\lg n)$.

Bài tập

13.4-1

Mô tả một cây tìm nhị phân trên n nút sao cho chiều sâu trung bình của một nút trong cây là $\Theta(\lg n)$ nhưng chiều cao của cây là $\omega(\lg n)$. Chiều cao của một cây tìm nhị phân n-nút có thể lớn đến mức bao nhiêu nếu chiều sâu trung bình của một nút là $\Theta(\lg n)$?

13.4-2

Chứng tỏ khái niệm của một cây tìm nhị phân được chọn ngẫu nhiên trên n khóa, ở đó mỗi cây tìm nhị phân gồm n khóa có khả năng như nhau để được chọn, sẽ khác với khái niệm của một cây tìm nhị phân xây dựng ngẫu nhiên đã cho trong đoạn này. (*Mách nước:* Liệt kê các khả năng khi $n = 3$.)

13.4-3 *

Cho một hằng $r \geq 1$, xác định t sao cho xác suất nhỏ hơn $1/n^r$ cho thấy chiều cao của một cây tìm nhị phân xây dựng ngẫu nhiên ít nhất là tHæ.

13.4-4 *

Xét RANDOMIZED-QUICKSORT hoạt động trên một dãy n con số nhập. Chứng minh với bất kỳ hằng $k > 0$, tất cả ngoại trừ $O(1 / n^k)$ của $n!$ phép hoán vị đầu vào đều cho ra một $O(n \lg n)$ thời gian thực hiện.

Các Bài Toán

13-1 Các cây tìm nhị phân có các khóa bằng nhau

Các khóa bằng nhau đặt ra một vấn đề cho việc thực thi các cây tìm nhị phân.

a. Đâu là khả năng thực hiện tiệm cận của TREE-INSERT khi được dùng để chèn n mục có các khóa đồng nhất vào một cây tìm nhị phân trống từ đầu?

Để cải thiện TREE-INSERT, chúng tôi đề xuất trắc nghiệm trước dòng 5 dẫu $key[z] = key[x]$ hay không và trắc nghiệm trước dòng 11 dẫu $key[z] = key[y]$ hay không. Nếu có đẳng thức, ta thực thi một trong các chiến lược dưới đây. Với mỗi chiến lược, bạn tìm khả năng thực hiện tiệm cận của việc chèn n mục có các khóa đồng nhất vào một cây tìm

nhị phân trống từ đầu. (Các chiến lược được mô tả cho dòng 5, ở đó ta so sánh các khóa của z và x. Dùng y thay cho x để đi đến các chiến lược cho dòng 11.)

b. Giữ một cờ bool $b[x]$ tại nút x, và ấn định x là *left*[x] hoặc *right*[x] dựa trên giá trị của $b[x]$, luân phiên giữa FALSE và TRUE mỗi lần nút được viếng thăm trong TREE-INSERT.

c. Giữ một danh sách các nút có các khóa bằng nhau tại x, và chèn z vào danh sách.

d. Ấn định ngẫu nhiên x là *left*[x] hoặc *right*[x]. (Nêu khả năng thực hiện trường hợp xấu nhất và, một cách không chính thức, suy ra khả năng thực hiện trường hợp trung bình.)

13-2 Các cây cơ số

Cho hai chuỗi $a = a_0a_1...a_p$ và $b = b_0b_1...b_q$, ở đó mỗi a_i và mỗi b_j nằm trong một tập hợp các ký tự nào đó có sắp xếp thứ tự, ta nói rằng chuỗi ***a theo từ điển nhỏ hơn*** chuỗi ***b*** nếu

1. Ở đó tồn tại một số nguyên j, $0 \leq j \leq \min(p, q)$, sao cho $a_i = b_i$ với tất cả

$i = 0,1,..., j - 1$ và $a_j < b_j$, hoặc

2. $p < q$ và $a_i = b_i$ với tất cả $i = 0, 1,..., p$.

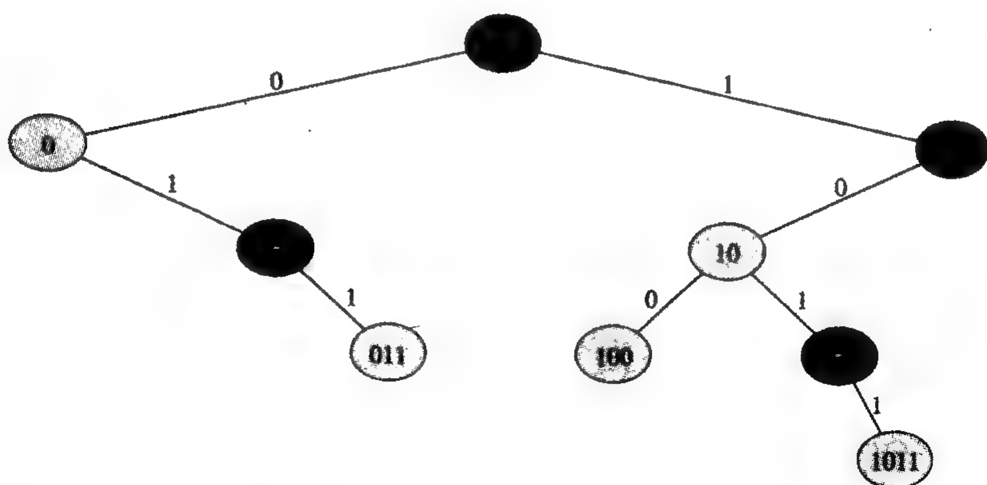
Ví dụ, nếu a và b là các chuỗi bit, thì 10100 < 10110 theo quy tắc 1 (cho $j = 3$) và 10100 < 101000 theo quy tắc 2. Điều này tương tự như cách sắp xếp thứ tự được dùng trong các từ điển tiếng Anh.

Cấu trúc dữ liệu ***cây cơ số*** nêu trong Hình 13.6 lưu trữ các chuỗi bit 1011, 10, 011, 100, và 0. Khi tìm kiếm một khóa $a = a_0a_1...a_p$, ta sang trái tại một nút có chiều sâu i nếu $a_i = 0$ và sang phải nếu $a_i = 1$. Cho S là một tập hợp các chuỗi nhị phân riêng biệt có các chiều dài tổng cộng tới n. Nêu cách sử dụng một cây cơ số để sắp xếp S theo kiểu từ điển trong $\Theta(n)$ thời gian. Với ví dụ trong Hình 13.6, kết xuất của đợt sắp xếp sẽ là dãy 0, 011, 10, 100, 1011.

13-3 Chiều sâu nút trung bình trong một cây tìm nhị phân xây dựng ngẫu nhiên

Trong bài toán này, ta chứng minh chiều sâu trung bình của một nút trong một cây tìm nhị phân xây dựng ngẫu nhiên có n nút là $O(\lg n)$. Mặc dù kết quả này yếu hơn so với Định lý 13.6, song kỹ thuật mà ta sẽ dùng sẽ nói lên một sự tương tự đáng ngạc nhiên giữa việc xây dựng một cây tìm nhị phân và tiến trình chạy RANDOMIZED-QUICKSORT từ Đoạn 8.3.

Để bắt đầu, chắc bạn còn nhớ, trong Chương 5 ta đã biết chiều dài lộ trình trong [internal path length] $P(T)$ của một cây nhị phân T chính là tổng, trên tất cả các nút x trong T, của chiều sâu của nút x, mà ta thể hiện bằng $d(x, T)$.



Hình 13.6 Một cây cơ số lưu trữ các chuỗi bit 1011, 10, 011, 100, và 0. Có thể xác định khóa của mỗi nút bằng cách băng ngang lộ trình từ gốc đến nút đó. Do đó không có nhu cầu lưu trữ các khóa trong các nút; các khóa được nêu ở đây chỉ để minh họa. Các nút được tô bóng đậm nếu các khóa tương ứng với chúng không nằm trong cây; các nút như vậy chỉ hiện diện để thiết lập một lộ trình đến các nút khác.

a. Chứng tỏ chiều sâu trung bình của một nút trong T là

$$\frac{1}{n}\sum_{x\in T} d(x, T) = \frac{1}{n}P(T) .$$

Như vậy, ta muốn chứng tỏ rằng giá trị dự trù của $P(T)$ là $O(n \lg n)$.

b. Cho T_L và T_R thể hiện các cây con trái và phải của cây T, theo thứ tự nêu trên.

Chứng tỏ nếu T có n nút, thì

$$P(T) = P(T_L) + P(T_R) + n - 1 .$$

c. Cho $P(n)$ thể hiện chiều dài lộ trình trong trung bình của một cây tìm nhị phân xây dựng ngẫu nhiên with n các nút. Chứng tỏ

$$P(n) = \frac{1}{n}\sum_{i=0}^{n-1} (P(i) + P(n-i-1) + n - 1) .$$

d. Chứng tỏ $P(n)$ có thể được viết lại là

$$P(n) = \frac{2}{n}\sum_{k=1}^{n-1} (P(k) + \Theta(n) .$$

e. Nhớ lại cuộc phân tích phiên bản ngẫu nhiên hóa của kỹ thuật sắp xếp nhanh, bạn hãy kết luận $P(n) = O(n \lg n)$.

Tại mỗi lần gọi đệ quy của kỹ thuật sắp xếp nhanh, ta chọn một thành phần trục [pivot element] ngẫu nhiên để phân hoạch tập hợp các thành phần được sắp xếp. Mỗi nút của một cây tìm nhị phân sẽ phân hoạch tập hợp các thành phần rơi vào cây con có gốc tại nút đó.

f. Mô tả một cách thực thi kỹ thuật sắp xếp nhanh ở đó các phép so sánh để sắp xếp một tập hợp các thành phần chính xác giống như các phép so sánh để chèn các thành phần vào một cây tìm nhị phân. (Thứ tự thực hiện các phép so sánh có thể khác nhau, nhưng phải thực hiện cùng các phép so sánh.)

13-4 Số cây nhị phân khác nhau

Cho b_n thể hiện số lượng cây nhị phân khác nhau có n nút. Trong bài toán này, bạn sẽ tìm một công thức cho b_n cũng như một ước lượng tiệm cận.

a. Chứng tỏ $b_0 = 1$ và chứng tỏ, với $n \geq 1$,

$$b_n = \sum_{k=0}^{n-1} b_k b_{n-1-k} .$$

b. Cho $B(x)$ là hàm phát sinh

$$B(x) = \sum_{n=0}^{\infty} b_n x^n$$

(xem Bài toán 4-6 để có phần định nghĩa về các hàm phát sinh). Chứng tỏ $B(x) = xB(x)^2 + 1$ và do đó

$$B(x) = \frac{1}{2x}(1 - \sqrt{1 - 4x}) .$$

Khai triển Taylor của $f(x)$ quanh điểm $x = a$ được gán bởi

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(x - a)}{k!} (x - a)^k ,$$

ở đó $f^{(k)}(x)$ là đạo hàm bậc k của f được đánh giá tại x.

c. Chứng tỏ

$$b_n = \frac{1}{n+1}\binom{2n}{n}$$

(***số Catalan*** [Catalan number] thứ n) bằng cách dùng phép khai triển Taylor của $\sqrt{1 - 4x}$ quanh $x = 0$. (Nếu muốn, thay vì dùng khai triển Taylor, bạn có thể dùng phép tổng quát hóa sự khai triển nhị thức (6.5) thành các số mũ không nguyên [nonintegral exponents] n, ở đó với bất kỳ số thực n và số nguyên k, ta diễn dịch $\binom{n}{k}$ là $n(n - 1)\ldots(n - k + 1)/k!$ nếu $k \geq 0$, và bằng không là 0.)

d. Chứng tỏ

$$b_n = \frac{4^n}{\sqrt{\pi} n^{3/2}} (1 + O(1/n)) .$$

Ghi chú Chương

Knuth [123] có một phần mô tả hay về các cây tìm nhị phân đơn giản và nhiều biến thể. Các cây tìm nhị phân dường như đã được phát hiện một cách độc lập bởi một số người vào cuối những năm 1950.

14 Các Cây Đỏ Đen

Chương 13 đã chứng tỏ một cây tìm nhị phân có chiều cao *h* có thể thực thi bất kỳ trong số các phép toán tập hợp động căn bản—như SEARCH, PREDECESSOR, SUCCESSOR, MINIMUM, MAXIMUM, INSERT, và DELETE—trong $O(h)$ thời gian. Như vậy, các phép toán tập hợp chạy nhanh nếu cây tìm có chiều cao nhỏ; nhưng nếu nó có chiều cao lớn, khả năng thực hiện của các phép toán có thể không tốt hơn so với một danh sách nối kết. Các cây đỏ đen là một trong số nhiều lược đồ cây tìm kiếm "cân bằng" để bảo đảm các phép toán tập hợp động căn bản chiếm $O(\lg n)$ thời gian trong trường hợp xấu nhất.

14.1 Các tính chất của các cây đỏ đen

Một ***cây đỏ đen*** [red-black tree] là một cây tìm nhị phân có một bit lưu trữ phụ trội mỗi nút: ***màu*** của nó, có thể là RED [đỏ] hoặc BLACK [đen]. Nhờ ràng buộc cách thức tô màu các nút [nodes] trên một lộ trình bất kỳ từ gốc đến một lá, các cây đỏ đen bảo đảm không có lộ trình nào dài hơn gấp đôi so với bất kỳ lộ trình nào khác, sao cho cây xấp xỉ ***cân bằng***.

Mỗi nút của cây giờ đây chứa các trường *color, key, left, right,* và *p*. Nếu một con hoặc cha của một nút không tồn tại, trường biến trỏ tương ứng của nút chứa giá trị NIL. Ta sẽ xem các NIL này như là các biến trỏ đến các nút ngoài (các lá) của cây tìm nhị phân và các nút mang khóa bình thường, như là các nút trong của cây.

Một cây tìm nhị phân là một cây đỏ đen nếu nó thỏa ***các tính chất đỏ đen*** dưới đây:

1. Mọi nút là đỏ hoặc đen.

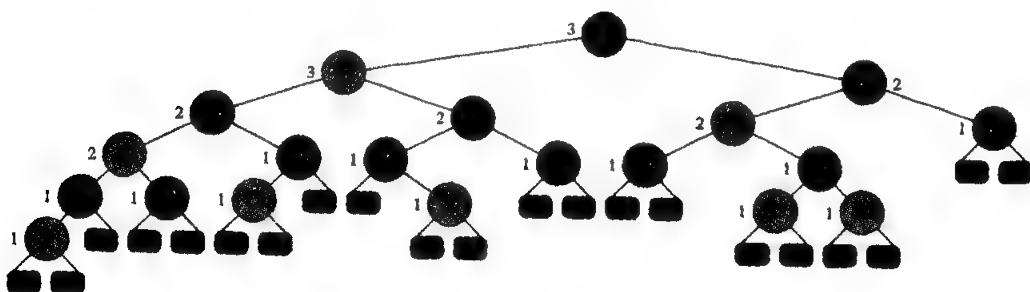
2. Mọi lá (NIL) là đen.

3. Nếu một nút đỏ, thì cả hai con của nó là đen.

4. Mọi lộ trình đơn giản từ một nút đến một lá hậu duệ chứa cùng số lượng nút đen.

Hình 14.1 có nêu ví dụ về một cây đỏ đen.

Ta gọi số lượng nút đen trên bất kỳ lộ trình nào từ (nhưng không gộp) một nút *x* đến một lá là ***chiều cao đen*** [black height] của nút, được biểu thị là bh(*x*). Theo tính chất 4, khái niệm của chiều cao đen được định nghĩa kỹ, bởi tất cả các lộ trình đi xuống từ nút đều có cùng số lượng các nút đen. Ta định nghĩa chiều cao đen của một cây đỏ đen là chiều cao đen của gốc của nó.



Hình 14.1 Một cây đỏ đen có các nút đen tô sẫm và các nút đỏ tô bóng. Mọi nút trong một cây đỏ đen là đỏ hoặc đen, mọi lá (NIL) là đen, các con của một nút đỏ đều đen, và mọi lộ trình đơn giản từ một nút đến một lá hậu duệ chứa cùng số lượng nút đen. Mỗi nút phi- NIL được đánh dấu bằng chiều cao đen của nó; có NIL có chiều cao đen là 0.

Bổ đề dưới đây cho biết tại sao các cây đỏ đen tạo các cây tìm kiếm tốt.

Bổ đề 14.1

Một cây đỏ đen với *n* nút trong [internal nodes] có chiều cao tối đa 2lg(*n* + 1).

Chứng minh Trước tiên ta chứng tỏ cây con có gốc tại một nút *x* bất kỳ sẽ chứa ít nhất $2^{bh(x)}$ - 1 nút trong. Ta chứng minh sự xác nhận này bằng phương pháp quy nạp trên chiều cao của *x*. Nếu chiều cao của *x* là 0, *x* phải là một lá (NIL), và như vậy cây con có gốc tại *x* chứa ít nhất $2^{bh(x)} - 1 = 2^0 - 1 = 0$ nút trong. Với bước quy nạp, ta xét một nút *x* có chiều cao dương và là một nút trong có hai con. Mỗi con có một chiều cao đen của bh(*x*) hoặc bh(*x*) - 1, tùy thuộc vào màu của nó là đỏ hoặc đen, theo thứ tự nêu trên. Bởi chiều cao của một con của *x* nhỏ hơn chiều cao của chính *x*, nên ta có thể áp dụng giả thuyết quy nạp để kết luận rằng mỗi con có ít nhất $2^{bh(x)-1}$ - 1 nút trong. Như vậy, cây con có gốc tại *x* chứa ít nhất $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ nút trong, chứng minh cho sự xác nhận.

Để hoàn chỉnh chứng minh của bổ đề, cho *h* là chiều cao của cây.

Theo tính chất 3, ít nhất phân nửa các nút trên một lộ trình đơn giản bất kỳ từ gốc đến một lá, không kể gốc, phải là đen. Do vậy, chiều cao đen của gốc phải ít nhất là $h/2$; như vậy,

$$n \geq 2^{h/2} - 1.$$

Việc dời 1 sang bên phía trái và lấy lôga trên cả hai phía sẽ cho ra $\lg(n + 1) \geq h/2$, hoặc $h \leq 2 \lg(n + 1)$.

Một hệ quả trực tiếp của bổ đề này đó là các phép toán tập hợp động SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, và PREDECESSOR có thể được thực thi trong $O(\lg n)$ thời gian trên các cây đỏ đen, bởi chúng có thể được thực hiện để chạy trong $O(h)$ thời gian trên một cây tìm có chiều cao h (như đã nêu trong Chương 13) và bất kỳ cây đỏ đen nào trên n nút đều là một cây tìm có chiều cao $O(\lg n)$. Mặc dù các thuật toán TREE-INSERT và TREE-DELETE ở Chương 13 chạy trong $O(\lg n)$ thời gian khi cho một cây đỏ đen làm nhập liệu, chúng không trực tiếp hỗ trợ các phép toán tập hợp động INSERT và DELETE, bởi chúng không bảo đảm cây tìm nhị phân đã sửa đổi sẽ là một cây đỏ đen. Tuy nhiên, như các Đoạn 14.3 và 14.4 cho thấy, hai phép toán này có thể được hỗ trợ trong $O(\lg n)$ thời gian.

Bài tập

14.1-1

Vẽ cây tìm nhị phân hoàn chỉnh có chiều cao 3 trên các khóa {1, 2,...,15}. Bổ sung các lá NIL và tô màu các nút theo ba cách khác nhau sao cho các chiều cao đen của các cây đỏ đen kết quả là 2, 3, và 4.

14.1-2

Giả sử gốc của một cây đỏ đen là đỏ. Nếu ta chuyển nó thành đen, cây có còn giữ nguyên là một cây đỏ đen hay không?

14.1-3

Chứng tỏ lộ trình đơn giản dài nhất từ một nút x trong một cây đỏ đen đến một lá hậu duệ có chiều dài tối đa gấp đôi so với lộ trình đơn giản ngắn nhất từ nút x đến một lá hậu duệ.

14.1-4

Đâu là số lượng khả dĩ lớn nhất của các nút trong trong một cây đỏ đen có chiều cao đen k? Đâu là số lượng khả dĩ nhỏ nhất?

14.1-5

Mô tả một cây đỏ đen trên n khóa thực hiện tỷ số khả dĩ lớn nhất của các nút đỏ trong với các nút đen trong. Tỷ số này là gì? Cây nào có tỷ số khả dĩ nhỏ nhất, và tỷ số đó là gì?

14.2 Các phép quay

Khi chạy trên một cây đỏ đen có *n* khóa, các phép toán cây tìm TREE-INSERT và TREE-DELETE mất $O(\lg n)$ thời gian. Do chúng sửa đổi cây, nên kết quả có thể vi phạm các tính chất đỏ-đen được nêu trong Đoạn 14.1. Để phục hồi các tính chất này, ta phải thay đổi các màu của vài nút trong cây và cũng thay đổi cấu trúc biến trỏ.

RIGHT-ROTATE(T, y)

LEFT-ROTATE(T, x)

Hình 14.2 Các phép toán quay trên một cây tìm nhị phân. Phép toán RIGHT-ROTATE(T,x) biến đổi cấu hình của hai nút bên trái thành cấu hình bên phải bằng cách thay đổi một số biến trỏ bất biến. Cấu hình bên phải có thể biến đổi thành cấu hình bên trái bằng phép toán nghịch đảo LEFT-ROTATE(T,y). Hai nút có thể xảy ra bất kỳ đâu trong một cây tìm nhị phân. Các mẫu tự α, β, và γ biểu thị cho các cây con tùy ý. Một phép quay bảo toàn cách sắp xếp thứ tự nội cấp [inorder] của các khóa: các khóa trong α đứng trước *key*[*x*], *key*[*x*] đứng trước các khóa trong β, các khóa này đứng trước *key*[*y*], *key*[*y*] lại đứng trước các khóa trong γ.

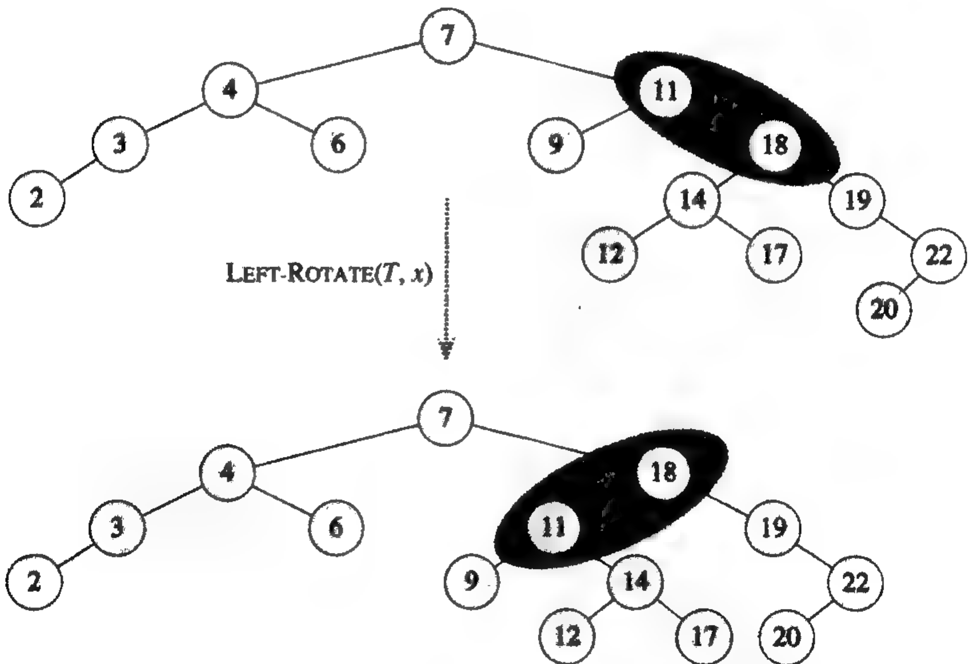
Ta thay đổi cấu trúc biến trỏ thông qua ***phép quay*** [rotation], là một phép toán cục bộ trong một cây tìm bảo toàn cách sắp xếp thứ tự khóa nội cấp. Hình 14.2 nêu hai kiểu phép quay: các phép quay trái và các phép quay phải. Khi thực hiện một phép quay trái trên một nút *x*, ta mặc nhận rằng con phải *y* của nó là phi-NIL. Phép quay trái "quay" quanh nối kết từ *x* đến *y*. Nó khiến *y* thành gốc mới của cây con, với *x* làm con trái của *y* và con trái của *y* làm con phải của *x*.

Mã giả của LEFT-ROTATE mặc nhận *right*[*x*] ≠ NIL.

```
LEFT-ROTATE (T, x)
1  y ← right[x]                ▷ Ấn định y.
2  right[x] ← left[y]          ▷ Chuyển cây con trái của y vào cây con
                                                  cây con phải của x .
3  if left[y] ≠ NIL
4      then p[left[y]] ← x
5  p[y] ← p[x]                 ▷ Nối kết cha của x với y.
```

```
if p[x] = NIL
    then root[T] ← y
    else if x = left[p[x]]
            then left[p[x]] ← y
            else right[p[x]] ← y
left[y] ← x            ▷ Đặt x bên trái của y .
p[x] ← y
```

Hình 14.3 nêu cách hoạt động của LEFT-ROTATE. Mã của RIGHT-ROTATE cũng tương tự. Cả LEFT-ROTATE lẫn RIGHT-ROTATE đều chạy trong $O(1)$ thời gian. Phép quay chỉ thay đổi các biến trỏ; tất cả các trường khác trong một nút vẫn giữ nguyên.



Hình 14.3 Một ví dụ về cách thức mà thủ tục LEFT-ROTATE(T,x) sửa đổi một cây tìm nhị phân. Các lá NIL được bỏ qua. Các di động cây nội cấp của cây nhập liệu và cây đã sửa đổi tạo ra cùng bảng kê các giá trị khóa.

Bài tập

14.2-1

Vẽ cây đỏ đen là kết quả sau khi gọi TREE-INSERT trên cây trong Hình 14.1 có khóa 36. Nếu nút chèn được tô màu đỏ, cây kết quả có

phải là một cây đỏ đen không? Điều gì xảy ra nếu nó được tô màu đen?

14.2-2

Viết mã giả cho RIGHT-ROTATE.

14.2-3

Chứng tỏ phép quay bảo toàn cách sắp xếp thứ tự khóa nội cấp của một cây nhị phân.

14.2-4

Cho a, b, và c là các nút tùy ý trong các cây con α, β, và γ, theo thứ tự nêu trên, trong cây trái của Hình 14.2. Các chiều sâu của a, b, và c thay đổi ra sao khi thực hiện một phép quay trái trên nút x trong hình?

14.2-5

Chứng tỏ có thể dùng $O(n)$ phép quay để biến đổi bất kỳ cây n-nút tùy ý nào thành một cây n-nút tùy ý bất kỳ khác. (*Mách nước*: Trước tiên chứng tỏ rằng tối đa n - 1 phép quay phải là đủ để biến đổi một cây bất kỳ thành chuỗi xích tiến phải.)

14.3 Phép chèn

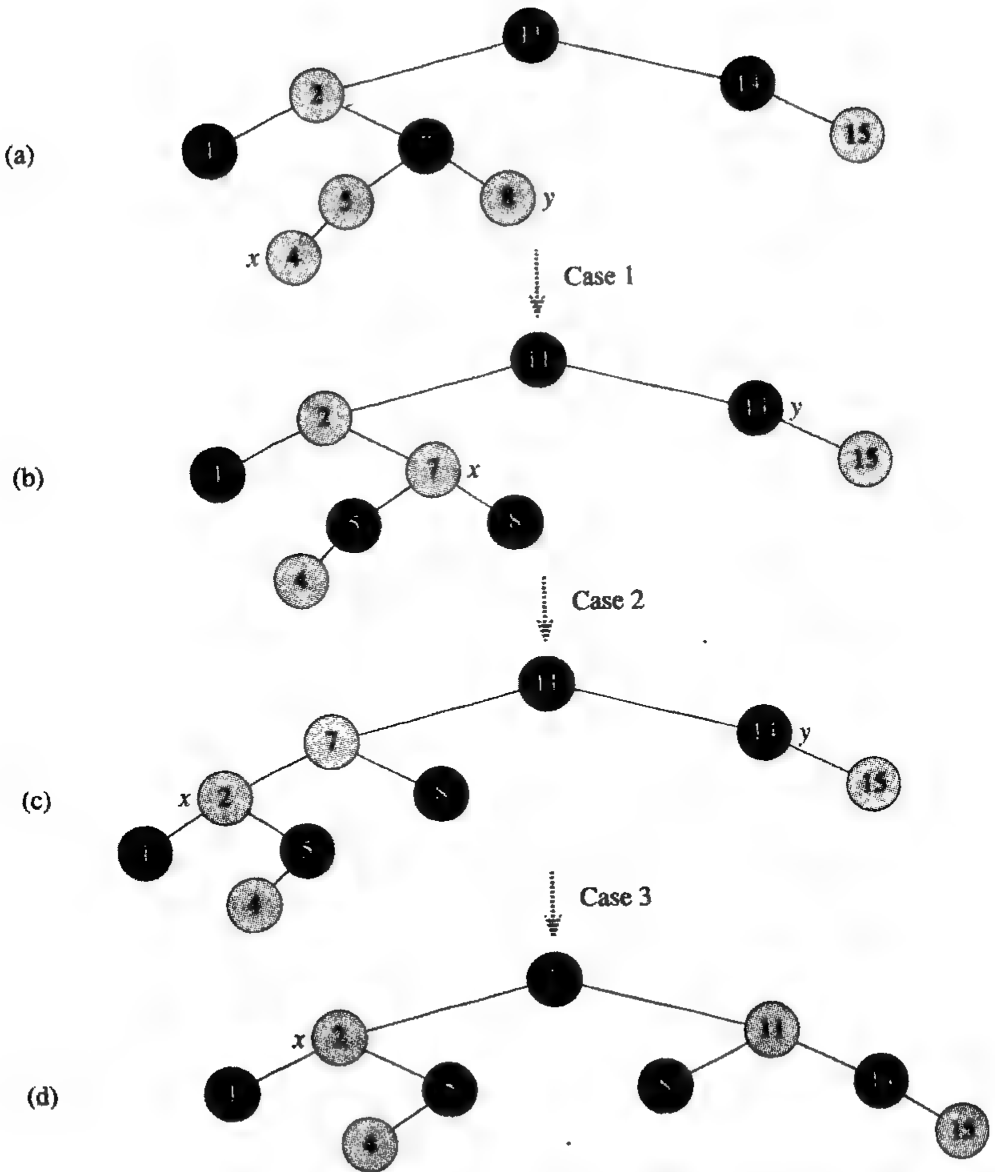
Có thể hoàn thành phép chèn một nút vào một cây đỏ đen n-nút trong $O(\lg n)$ thời gian. Ta dùng thủ tục TREE-INSERT (Đoạn 13.3) để chèn nút x vào cây T như thể nó là một cây tìm nhị phân bình thường, rồi ta tô đỏ x. Để bảo toàn các tính chất đỏ-đen, ta chỉnh sửa cây đã sửa đổi bằng cách tô màu lại các nút và thực hiện các phép quay. Phần lớn mã của RB-INSERT điều quản các trường hợp khác nhau có thể nảy sinh khi ta chỉnh sửa cây đã sửa đổi.

```
RB-INSERT( T, x )
TREE-INSERT(T,x)
color[x] ← RED
while x ≠ root[T] and color[p[x]] = RED
     do if  p[x] = left[p[p[x]]]
           then y ← right[p[p[x]]]
                if  color[y] = RED
```

```
7            then color[p[x]] ← BLACK          ▷ Trường hợp 1
8                 color[y] ← BLACK             ▷ Trường hợp 1
9                 color[p[p[x]]] ← RED         ▷ Trường hợp 1
10                x ← p[p[x]]                  ▷ Trường hợp 1
11           else if x = right[p[x]]
12                then x ← p[x]                ▷ Trường hợp 2
13                     LEFT-ROTATE(T,x)
                                               ▷ Trường hợp 2
14                color[p[x]] ← BLACK          ▷ Trường hợp 3
15                color[p[p[x]]] ← RED         ▷ Trường hợp 3
16                RIGHT-ROTATE(T,p[p[x]])
                                               ▷ Trường hợp 3
17           else (giống như mệnh đề
                  then với "right" và "left" trao đổi)
18 color[root[T]] ← BLACK
```

Mã của RB-INSERT không gây ấn tượng mạnh như dáng vẻ bề ngoài của nó. Ta sẽ xem xét mã theo ba bước chính. Trước hết, ta xác định các vi phạm đối với các tính chất đỏ-đen xuất hiện trong các dòng 1-2 khi nút *x* được chèn và được tô màu đỏ. Thứ hai, ta xem xét mục tiêu chung của vòng lặp **while** trong các dòng 3-17. Cuối cùng, ta khảo sát mỗi trong số ba trường hợp mà vòng lặp **while** được tách và xem cách chúng hoàn thành mục tiêu ra sao. Hình 14.4 nêu cách hoạt động của RB-INSERT trên một cây đỏ đen mẫu.

Tính chất đỏ-đen nào có thể bị vi phạm sau các dòng 1-2? Chắc chắn là tính chất 1 tiếp tục duy trì, cũng như tính chất 2, bởi nút đỏ mới được chèn có các NIL cho các con. Tính chất 4 nói rằng số lượng các nút đen là giống nhau trên mọi lộ trình từ một nút đã cho, và nó cũng được thỏa, bởi vì nút *x* thay một NIL (đen), và nút *x* là đỏ với các con NIL. Như vậy, tính chất duy nhất có thể bị vi phạm là tính chất 3, nói rằng một nút đỏ không thể có một con đỏ. Cụ thể, tính chất 3 bị vi phạm nếu cha của *x* là đỏ, bởi chính *x* được tô đỏ trong dòng 2. Hình 14.4(a) nêu một kiểu vi phạm như vậy sau khi chèn nút *x*.



Hình 14.4 Phép toán của RB-INSERT. **(a)** Một nút x sau khi chèn. Bởi x và cha của nó $p[x]$ cả hai đều là đỏ, một sự vi phạm đối với tính chất 3 xảy ra. Do bác y của x là đỏ, ta có thể áp dụng trường hợp 1 trong mã. Các nút được tô màu lại và biến trỏ x được dời lên cây, dẫn đến cây được nêu trong **(b)**. Một lần nữa, x và cha của nó cả hai đều có màu đỏ, nhưng bác y của x là đen. Do x là con phải của $p[x]$, nên trường hợp 2 có thể được áp dụng. Một phép quay trái được thực hiện, và cây kết quả được nêu trong **(c)**. Giờ đây x là con trái của cha nó, và trường hợp 3 có thể được áp dụng. Một phép quay phải cho ra cây trong **(d)**, là một cây đỏ đen hợp pháp.

Mục tiêu của vòng lặp **while** trong các dòng 3-17 đó là dời vi phạm đối với tính chất 3 lên phía trên cây trong khi duy trì tính chất 4 dưới dạng một bất biến. Tại đầu mỗi lần lặp lại của vòng lặp, x trỏ đến một nút đỏ với một cha đỏ—vi phạm duy nhất trong cây. Có hai kết quả khả dĩ cho mỗi lần lặp lại của vòng lặp: biến trỏ x dời lên trên cây, hoặc thực hiện vài phép quay và vòng lặp kết thúc.

Thực tế có sáu trường hợp để xét trong vòng lặp **while**, nhưng ba trong số chúng là đối xứng với ba trường hợp kia, tùy thuộc vào việc cha $p[x]$ của x là một con trái hay con phải của ông nội $p[p[x]]$ của x, được xác định trong dòng 4. Ta chỉ gán mã cho tình huống ở đó $p[x]$ là một con trái. Ta đã thực hiện giả thiết quan trọng rằng gốc của cây là đen—một tính chất mà ta bảo đảm trong dòng 18 mỗi lần kết thúc—sao cho $p[x]$ không phải là gốc và $p[p[x]]$ tồn tại.

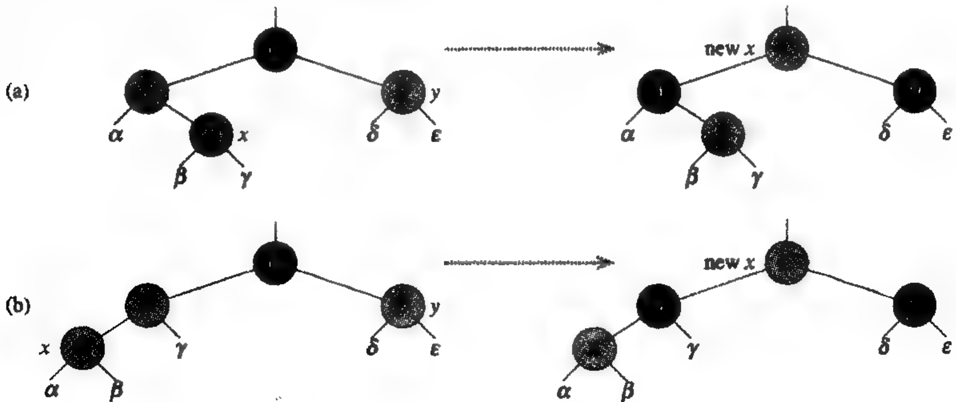
Trường hợp 1 được phân biệt với các trường hợp 2 và 3 bằng màu của anh em song sinh của cha của x, tức "bác." Dòng 5 khiến y trỏ đến bác $right[p[p[x]]]$ của x, và một đợt trắc nghiệm được thực hiện trong dòng 6. Nếu y là đỏ, thì trường hợp 1 được thi hành. Bằng không, quyền điều kiện chuyển về cho các trường hợp 2 và 3. Trong cả ba trường hợp, ông nội $p[p[x]]$ của x là đen, bởi cha $p[x]$ của nó là đỏ, và tính chất 3 chỉ bị vi phạm giữa x và $p[x]$.

Tình huống của trường hợp 1 (các dòng 7-10) được nêu trong Hình 14.5. Trường hợp 1 được thi hành khi cả $p[x]$ lẫn y là đỏ. Bởi $p[p[x]]$ là đen, ta có thể tô màu cả $p[x]$ lẫn y là đen, nhờ đó chỉnh sửa sự cố của x và $p[x]$ cả hai đều là đỏ, và tô màu cho $p[p[x]]$ là đỏ, nhờ đó duy trì tính chất 4. Sự cố duy nhất có thể nảy sinh đó là $p[p[x]]$ có thể có một cha đỏ; do đó, ta phải lặp lại vòng lặp **while** với $p[p[x]]$ làm nút x mới.

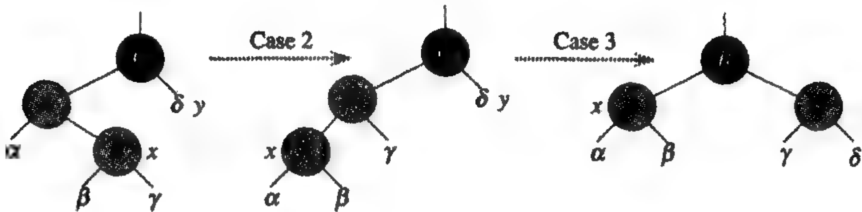
Trong các trường hợp 2 và 3, màu của bác y của x là đen. Hai trường hợp được phân biệt bằng x là một con phải hay con trái của $p[x]$. Các dòng 12-13 tạo thành trường hợp 2, được nêu trong Hình 14.6 chung với trường hợp 3. Trong trường hợp 2, nút x là một con phải của cha của nó. Ta lập tức dùng một phép quay trái để biến đổi tình huống thành trường hợp 3 (các dòng 14-16), ở đó nút x là một con trái. Do cả x lẫn $p[x]$ đều đỏ, nên phép quay không ảnh hưởng đến chiều cao đen của các nút cũng như tính chất 4. Dẫu ta trực tiếp nhập trường hợp 3 hoặc thông qua trường hợp 2, bác y của x vẫn là đen, bởi bằng không ta đã thi hành trường hợp 1. Ta thi hành vài thay đổi màu và một phép quay phải để bảo toàn tính chất 4, sau đó, bởi không còn có hai nút đỏ trong một hàng, nên ta đã hoàn tất. Thân của vòng lặp **while** không được thi hành một lần khác, bởi $p[x]$ giờ đây là đen.

Đâu là thời gian thực hiện của RB-INSERT? Do chiều cao của một cây đỏ đen trên n nút là $O(\lg n)$, nên lệnh gọi đến TREE-INSERT sẽ mất

$O(\lg n)$ thời gian. Vòng lặp **while** chỉ lặp lại nếu trường hợp 1 được thi hành, sau đó biến trỏ x dời lên cây. Do đó, tổng số lần mà vòng lặp **while** có thể được thi hành là $O(\lg n)$. Như vậy, RB-INSERT mất tổng cộng là $O(\lg n)$ thời gian. Điều thú vị đó là, nó không bao giờ thực hiện trên hai phép quay, bởi vòng lặp **while** kết thúc nếu trường hợp 2 hoặc trường hợp 3 được thi hành.



Hình 14.5 Trường hợp 1 của thủ tục RB-INSERT. Tính chất 3 bị vi phạm, bởi cả x lẫn cha $p[x]$ của nó đều màu đỏ. Cùng hành động được thực hiện dẫu **(a)** x là một con phải hoặc **(b)** x là một con trái. Mỗi cây con α, β, γ, δ, và ε đều có một gốc đen, và mỗi cây con đều có cùng chiều cao đen. Mã của trường hợp 1 sẽ thay đổi các màu của vài nút, bảo toàn được tính chất 4: tất cả các lộ trình đi xuống từ một nút đến một lá đều có cùng số lượng nút đen. Vòng lặp **while** tiếp tục với ông nội $p[p[x]]$ của nút x làm x mới. Mọi vi phạm đối với tính chất 3 giờ đây chỉ có thể xảy ra giữa x mới, là đỏ, và cha của nó, nếu nó cũng màu đỏ.



Hình 14.6 Các trường hợp 2 và 3 của thủ tục RB-INSERT. Như trong trường hợp 1, tính chất 3 bị vi phạm trong trường hợp 2 hoặc trường hợp 3 bởi x và cha $p[x]$ của nó đều mang màu đỏ. Mỗi cây con α, β, γ, và δ đều có một gốc đen, và mỗi cây con có cùng chiều cao đen. Trường hợp 2 được biến đổi thành trường hợp 3 bằng một phép quay trái để bảo toàn tính chất 4: tất cả các lộ trình đi xuống từ một nút đến một lá đều có cùng số lượng nút đen. Trường hợp 3 gây ra vài thay đổi màu và một phép quay phải, cũng bảo toàn tính chất 4. Sau đó, vòng lặp **while** kết thúc, bởi tính chất 3 được thỏa: không còn hai nút đỏ trong một hàng.

Bài tập

14.3-1

Trong dòng 2 của RB-INSERT, ta ấn định màu của nút x mới được chèn thành đỏ. Lưu ý, nếu ta đã chọn để ấn định màu của x là đen, thì tính chất 3 của một cây đỏ đen sẽ không bị vi phạm. Tại sao ta không chọn ấn định màu của x là đen?

14.3-2

Trong dòng 18 của RB-INSERT, ta ấn định màu của gốc là đen. Nêu ưu điểm khi làm thế?

14.3-3

Nêu các cây đỏ đen là kết quả sau khi chèn thành công các khóa 41, 38, 31, 12, 19, 8 vào một cây đỏ đen trống từ đầu.

14.3-4

Giả sử chiều cao đen của mỗi cây con α, β, γ, δ, ε trong các Hình 14.5 và 14.6 là k. Gán nhãn mỗi nút trong từng hình với chiều cao đen của nó để xác minh tính chất 4 được bảo toàn bởi sự biến đổi đã chỉ.

14.3-5

Xét một cây đỏ đen được tạo dựng bằng cách chèn n nút với RB-INSERT. Chứng tỏ nếu $n > 1$, cây sẽ có ít nhất một nút đỏ.

14.3-6

Gợi ý cách thực thi RB-INSERT hiệu quả nếu phần biểu diễn của các cây đỏ đen không gộp kho lưu trữ cho các biến trỏ cha.

14.4 Phép xóa

Cũng như các phép toán căn bản khác trên một cây đỏ đen n-nút, phép xóa của một nút mất một thời gian $O(\lg n)$. Xóa một nút ra khỏi một cây đỏ đen chỉ hơi phức tạp hơn tiến trình chèn một nút chút đỉnh.

Để rút gọn các điều kiện cận trong mã, ta dùng một cờ hiệu để biểu diễn NIL (xem trang 229). Với một cây đỏ đen T, cờ hiệu $nil[T]$ là một đối tượng có các trường tương tự như một nút bình thường trong cây. Trường *color* của nó là BLACK, và các trường khác của nó—*p*, *left*, *right*, và *key*—có thể được ấn định theo các giá trị tùy ý. Trong cây đỏ đen, tất cả các biến trỏ đến NIL đều được thay bởi các biến trỏ đến cờ hiệu $nil[T]$.

Ta dùng các cờ hiệu để có thể xử lý một con NIL của một nút x như

một nút bình thường có cha là *x*. Có thể bổ sung một nút cờ hiệu riêng biệt cho từng NIL trong cây, sao cho cha của mỗi NIL cũng được định nghĩa, nhưng điều đó sẽ làm lãng phí không gian. Thay vì thế, ta dùng cờ hiệu duy nhất *nil* [*T*] để biểu diễn tất cả các NIL. Tuy nhiên, khi muốn điều tác một con của một nút *x*, ta phải cẩn thận ấn định *p*[*nil*[*T*]] theo *x* trước.

Thủ tục RB-DELETE là một sửa đổi nhỏ của thủ tục TREE-DELETE (Đoạn 13.3). Sau khi nối khử một nút, nó gọi một thủ tục phụ RB-DELETE-FIXUP thay đổi các màu và thực hiện các phép quay để phục hồi các tính chất đỏ-đen.

```
RB-DELETE(T, z)
 1 if   left[z] = nil[T] or right[z] = nil[T]
 2        then y ← z
 3        else y ← TREE-SUCCESSOR(z)
 4 if left[y] ≠ nil[T]
 5        then x ← left[y]
 6        else x ← right[y]
 7 p[x] ← p[y]
 8 if p[y] = nil[T]
 9        then root[T] ← x
10        else if y = left[p[y]]
11                 then left[p[y]] ← x
12                 else right[p[y]] ← x
13 if y ≠ z
14        then key[z] ← key[y]
15                 ▷ If y có các trường khác, chép chúng copy chúng, too.
16 if color[y] = BLACK
17        then RB-DELETE-FIXUP(T, x)
18 return y
```

Có ba điểm khác nhau giữa các thủ tục TREE-DELETE và RB-DELETE. Trước hết, tất cả các tham chiếu đến NIL trong TREE-DELETE đều đã được thay bằng các tham chiếu đến cờ hiệu *nil*[*T*] trong RB-DELETE. Thứ hai, đợt trắc nghiệm xem *x* có phải là NIL hay không trong dòng 7 của TREE-DELETE đã được gỡ bỏ, và phép gán *p*[*x*] - *p*[*y*] được thực hiện vô điều kiện trong dòng 7 của RB-DELETE. Như vậy,

nếu x là cờ hiệu *nil*[*T*], biến trỏ cha của nó trỏ đến cha của nút y đã được nối khử. Thứ ba, một lệnh gọi đến RB-DELETE-FIXUP được thực hiện trong các dòng 16-17 nếu y là đen. Nếu y là đỏ, các tính chất đỏ-đen vẫn được duy trì khi y được nối khử, bởi không có chiều cao đen nào trong cây đã thay đổi và không có nút đỏ nào đã được chuyển thành kề. Nút x được chuyền cho RB-DELETE-FIXUP là nút mà trước đó là con một của y trước khi y được nối khử nếu y có một con phi NIL, hoặc là cờ hiệu *nil*[*T*] nếu y không có các con. Trong trường hợp sau, phép gán vô điều kiện trong dòng 7 bảo đảm cha của x giờ đây là nút mà trước đó là cha của y, dẫu x là một nút trong mang khóa hoặc cờ hiệu *nil*[*T*].

Giờ đây ta có thể xem xét cách thức mà thủ tục RB-DELETE-FIXUP phục hồi các tính chất đỏ-đen đối với cây tìm.

```
RB-DELETE-FIXUP(T, x)
1   while x ≠ root[T] và color[x] = BLACK
2       do if x = left[p[x]]
3           then w ← right[p[x]]
4                if color[w] = RED
5                   then color[w] ← BLACK            ▷ Trường hợp 1
6                        color[p[x]] ← RED           ▷ Trường hợp 1
7                        LEFT-ROTATE(T,p[x])         ▷ Trường hợp 1
8                        w ← right[p[x]]             ▷ Trường hợp 1
9                if color[left[w ]] = BLACK
                             và color[right[w ]] = BLACK
10                  then color[w ] ← RED             ▷ Trường hợp 2
11                       x ← p[x]                    ▷ Trường hợp 2
12                  else if color[right[w]] = BLACK
13                       then color[left[w]] ← BLACK
                                                     ▷ Trường hợp 3
14                       color[w] ← RED              ▷ Trường hợp 3
15                       RIGHT-ROTATE(T,w)           ▷ Trường hợp 3
16                       w ←right[p[x]]              ▷ Trường hợp 3
17                       color[w] ← color[p[x]]      ▷ Trường hợp 4
18                       color[p[x]] ← BLACK         ▷ Trường hợp 4
```

```
19                      color[right[w]] , BLACK  ▷ Trường hợp 4
20                      LEFT-ROTATE(T,p[x])      ▷Trường hợp 4
21                      x ← root[T]              ▷Trường hợp 4
22              else (giống như mệnh đề
                        then với "right" và "left" trao đổi)
23  color[x] ← BLACK
```

Nếu nút bị nối khử *y* trong RB-DELETE là đen, việc gỡ bỏ nó sẽ khiến bất kỳ lộ trình nào mà trước đó chứa nút *y* có ít hơn một nút đen. Như vậy, tính chất 4 giờ đây bị vi phạm bởi bất kỳ tiền bối nào của *y* trong cây. Ta có thể chỉnh sửa sự cố bằng cách coi nút *x* như thể có một nút đen "phụ trội." Nghĩa là, nếu cộng 1 vào số đếm các nút đen trên bất kỳ lộ trình nào chứa *x*, thì theo sự diễn dịch này, tính chất 4 vẫn duy trì. Khi nối khử nút đen *y*, ta "đẩy" tính đen của nó lên trên con của nó. Vấn đề duy nhất đó là giờ đây nút *x* có thể là "đen kép," do đó vi phạm tính chất 1.

Thủ tục RB-DELETE-FIXUP gắng phục hồi tính chất 1. Mục tiêu của vòng lặp **while** trong các dòng 1-22 đó là dời nút đen phụ trội lên cây cho đến khi (1) *x* trỏ đến một nút đỏ, trong trường hợp đó ta tô màu nút đen trong dòng 23, (2) *x* trỏ đến gốc, trong trường hợp đó nút đen phụ trội có thể đơn giản "được gỡ bỏ," hoặc (3) có thể thực hiện các phép quay và tô màu lại cho phù hợp.

Bên trong vòng lặp **while**, *x* luôn trỏ đến một nút đen phi gốc có nút đen phụ trội. Trong dòng 2 ta xác định *x* là một con trái hay là một con phải của cha *p*[*x*] của nó. (Ta đã gán mã cho tình huống ở đó *x* là một con trái; tình huống ở đó *x* là một con phải—dòng 22—là đối xứng.) Ta duy trì một biến trỏ *w* đến anh em song sinh của *x*. Bởi nút *x* đen kép, nên nút *w* không thể là *nil*[*T*]; bằng không, số lượng đen trên lộ trình từ *p*[*x*] đến lá NIL *w* sẽ nhỏ hơn số lượng trên lộ trình từ *p*[*x*] đến *x*.

Hình 14.7 minh họa bốn trường hợp trong mã. Trước khi xem xét chi tiết từng trường hợp, ta hãy xem khái quát cách để xác minh việc biến đổi trong mỗi trường hợp sẽ bảo toàn tính chất 4. Ý tưởng chính đó là trong mỗi trường hợp, tiến trình biến đổi sẽ bảo toàn số lượng nút đen từ (và kể cả) gốc của cây con được nêu cho từng cây con α, β,..., ζ. Ví dụ, trong Hình 14.7(**a**), minh họa trường hợp 1, số lượng nút đen từ gốc đến cây con α hoặc β là 3, cả trước lẫn sau khi biến đổi. (Nên nhớ, biến

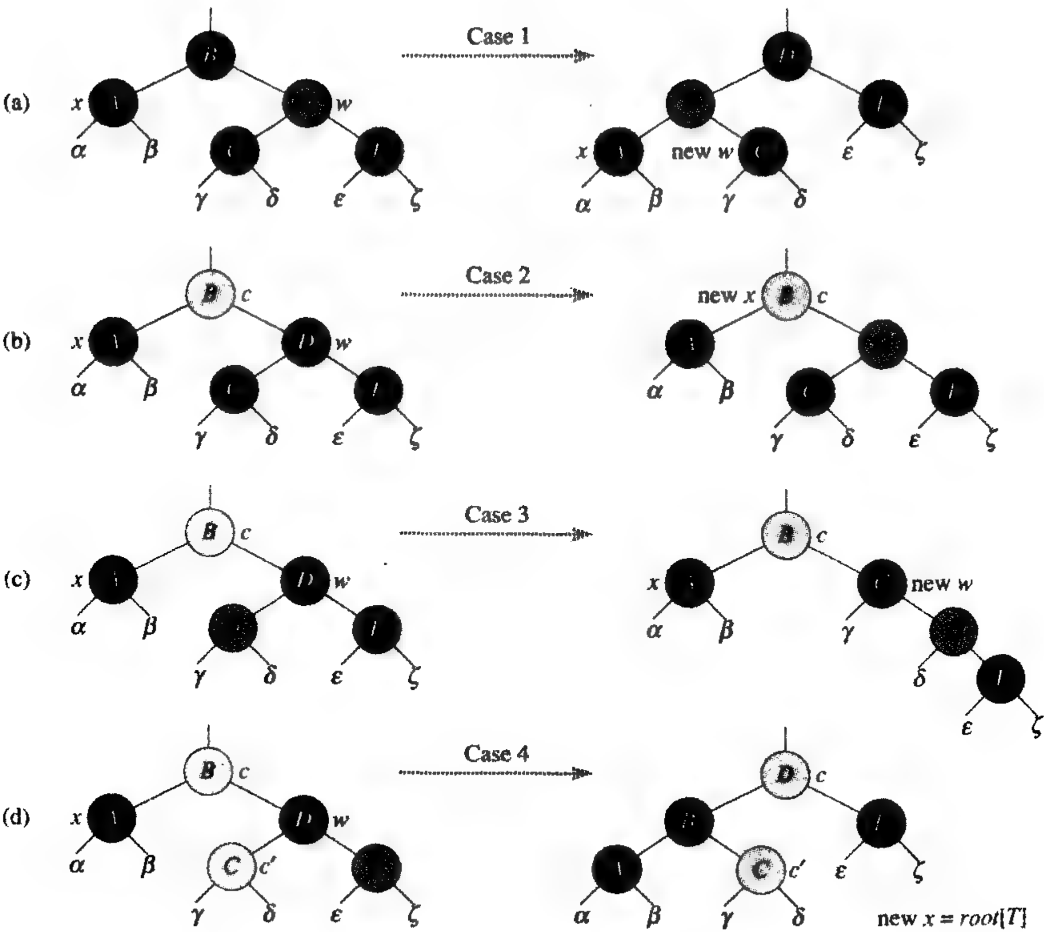
trỏ x bổ sung một nút đen phụ trội.) Cũng vậy, số lượng nút đen từ gốc đến bất kỳ trong số γ, δ, ε, và ζ là 2, cả trước lẫn sau khi biến đổi. Trong Hình 14.7(b), tiến trình đếm phải liên quan đến màu c, có thể là đỏ hoặc đen. Nếu ta định nghĩa count(RED) = 0 và count(BLACK) = 1, thì số lượng nút đen từ gốc đến α là 2 + count(c), cả trước lẫn sau biến đổi. Các trường hợp khác có thể được xác minh tương tự (Bài tập 14.4-5).

Trường hợp 1 (các dòng 5-8 của RB-DELETE-FIXUP và Hình 14.7(a)) xảy ra khi nút w, anh em song sinh của nút x, là đỏ. Bởi w phải có các con đen, nên ta có thể chuyển các màu của w và $p[x]$ rồi thực hiện một phép quay trái trên $p[x]$ mà không vi phạm bất kỳ tính chất đỏ-đen nào. Anh em song sinh mới của x, một trong các các con của w, giờ đây là đen, và như vậy ta đã chuyển đổi trường hợp 1 thành trường hợp 2, 3, hoặc 4.

Các trường hợp 2, 3, và 4 xảy ra khi nút w là đen; chúng được phân biệt bằng các màu của các con của w. Trong trường hợp 2 (các dòng 10-11 của RB-DELETE-FIXUP và Hình 14.7(b)), cả hai con của w là đen. Bởi w cũng đen, nên ta khử một nút đen ra khỏi cả x lẫn w, chỉ để x với nút đen và để w đỏ, và bổ sung một nút đen phụ trội vào $p[x]$. Sau đó, lặp lại vòng lặp **while** với $p[x]$ làm nút x mới. Nhận thấy nếu nhập trường hợp 2 qua trường hợp 1, màu c của nút x mới là đỏ, bởi $p[x]$ ban đầu là đỏ, và như vậy vòng lặp kết thúc khi nó trắc nghiệm điều kiện vòng lặp.

Trường hợp 3 (các dòng 13-16 và Hình 14.7(c)) xảy ra khi w là đen, con trái của nó là đỏ, và con phải của nó là đen. Ta có thể chuyển các màu của w và con trái $left[w]$ của nó rồi thực hiện một phép quay phải trên w mà không vi phạm các tính chất đỏ-đen. Anh em song sinh mới w của x giờ đây là một nút đen có một con phải đỏ, và như vậy ta đã biến đổi trường hợp 3 thành trường hợp 4.

Trường hợp 4 (các dòng 17-21 và Hình 14.7(d)) xảy ra khi anh em song sinh w của nút x là đen và con phải của w là đỏ. Nhờ thực hiện vài thay đổi màu và thực hiện một phép quay trái trên $p[x]$, ta có thể gỡ bỏ nút đen phụ trội trên x mà không vi phạm các tính chất đỏ-đen. Việc ấn định x là gốc sẽ khiến vòng lặp **while** kết thúc khi nó trắc nghiệm điều kiện vòng lặp.



Hình 14.7 Các trường hợp trong vòng lặp **while** của thủ tục RB-DELETE. Các nút sẫm là đen, các nút tô bóng đậm là đỏ, và các nút tô bóng sáng, có thể là đỏ hoặc đen, được biểu thị bởi c và c'. Các mẫu tự $\alpha, \beta, ..., \zeta$ biểu diễn các cây con tùy ý. Trong mỗi trường hợp, cấu hình bên trái được biến đổi thành cấu hình bên phải bằng cách thay đổi vài màu và/hoặc thực hiện một phép quay. Một nút được x trỏ đến sẽ có một nút đen phụ trội. Trường hợp duy nhất khiến vòng lặp lại đó là trường hợp 2. **(a)** Trường hợp 1 được biến đổi thành trường hợp 2, 3, hoặc 4 bằng cách trao đổi các màu của các nút B và D và thực hiện một phép quay trái. **(b)** Trong trường hợp 2, nút đen phụ trội mà biến trỏ x biểu thị được dời lên cây bằng cách tô màu nút D đỏ và ấn định x để trỏ đến nút B. Nếu ta nhập trường hợp 2 qua trường hợp 1, vòng lặp **while** kết thúc, bởi màu c là đỏ. **(c)** Trường hợp 3 được biến đổi thành trường hợp 4 bằng cách trao đổi các màu của các nút C và D và thực hiện một phép quay phải. **(d)** Trong trường hợp 4, nút đen phụ trội được biểu diễn bởi x có thể được gỡ bỏ bằng cách thay đổi vài màu và thực hiện một phép quay trái (mà không vi phạm các tính chất đỏ-đen), và vòng lặp kết thúc.

Nêu thời gian thực hiện của RB-DELETE? Do chiều cao của một cây đỏ-đen có *n* nút là $O(\lg n)$, nên tổng hao phí của thủ tục mà không có lệnh gọi đến RB-DELETE-FIXUP chiếm $O(\lg n)$ thời gian. Trong RB-DELETE-FIXUP, từng trường hợp 1, 3, và 4 sẽ kết thúc sau khi thực hiện một số thay đổi màu bất biến và tối đa là ba phép quay. Trường hợp 2 là trường hợp duy nhất ở đó có thể lặp lại vòng lặp **while**, rồi biến trỏ *x* dời lên cây tối đa $O(\lg n)$ thời gian và không thực hiện phép quay nào. Như vậy, thủ tục RB-DELETE-FIXUP mất $O(\lg n)$ thời gian và thực hiện tối đa ba phép quay, và do đó thời gian chung của RB-DELETE cũng là $O(\lg n)$.

Bài tập

14.4-1

Chứng tỏ gốc của cây đỏ đen luôn đen sau khi RB-DELETE thi hành.

14.4-2

Trong Bài tập 14.3-3, bạn đã thấy cây đỏ đen là kết quả của tiến trình chèn thành công các khóa 41, 38, 31, 12, 19, 8 vào một cây trống từ đầu. Giờ đây, chứng tỏ các cây đỏ đen, là kết quả từ phép xóa kế tiếp của các khóa theo thứ tự 8, 12, 19, 31, 38, 41.

14.4-3

Ta có thể xem xét hoặc sửa đổi cờ hiệu *nil*[*T*] trong những dòng nào của mã viết cho RB-DELETE-FIXUP?

14.4-4

Hãy đơn giản hóa mã của LEFT-ROTATE bằng cách dùng một cờ hiệu cho NIL và một cờ hiệu khác để lưu giữ biến trỏ đến gốc.

14.4-5

Trong mỗi trường hợp của Hình 14.7, ta cho số đếm các nút đen từ gốc của cây con được nêu cho từng cây con $\alpha, \beta, \ldots, \zeta$, và xác minh mỗi số đếm vẫn giữ nguyên sau khi biến đổi. Khi một nút có một màu *c* hoặc *c'*, hãy dùng hệ ký hiệu count(*c*) hoặc count(*c'*) trong số đếm của bạn.

14.4-6

Giả sử một nút *x* được chèn vào một cây đỏ đen bằng RB-INSERT rồi lập tức được xóa bằng RB-DELETE. Cây đỏ đen kết quả có giống như cây đỏ đen ban đầu không? Xác minh câu trả lời.

Các Bài Toán

14-1 Các tập hợp động bền

Trong quá trình diễn tiến của một thuật toán, đôi lúc ta thấy cần duy trì các phiên bản quá khứ của một tập hợp động khi nó được cập nhật. Kiểu tập hợp như vậy được gọi là ***bền*** [persistent]. Một cách để thực thi một tập hợp bền đó là chép nguyên cả tập hợp mỗi khi nó được sửa đổi, nhưng cách tiếp cận này có thể làm chậm một chương trình và cũng tiêu thụ nhiều không gian. Đôi lúc, ta có thể thực hiện tốt hơn nhiều.

Xét một tập hợp bền S có các phép toán INSERT, DELETE, và SEARCH, mà ta thực thi bằng các cây tìm nhị phân như đã nêu trong Hình 14.8(a). Ta duy trì một gốc riêng biệt cho mọi phiên bản của tập hợp. Để chèn khóa 5 vào tập hợp, ta tạo một nút mới có khóa 5. Nút này trở thành con trái của một nút mới có khóa 7, bởi ta không thể sửa đổi nút hiện có khóa 7. Cũng vậy, nút mới có khóa 7 sẽ trở thành con trái của một nút mới có khóa 8 mà con phải của nó là nút hiện có khóa 10. Đến lượt nút mới có khóa 8 trở thành con phải của một gốc mới r' có khóa 4 mà con trái của nó là nút hiện có khóa 3. Như vậy ta chỉ chép một phần của cây và chia sẻ vài trong số các nút có cây ban đầu, như đã nêu trong Hình 14.8(b).

a. Với một cây tìm nhị phân bền tổng quát [general persistent binary search tree], hãy định danh các nút cần được thay đổi để chèn một khóa k hoặc xóa một nút y.

b. Viết một thủ tục PERSISTENT-TREE-INSERT để , căn cứ vào một cây bền T và một khóa k để chèn, trả về một cây bền T' mới, là kết quả của việc chèn k vào T. Giả sử mỗi cây nút có các trường *key*, *left*, và *right* nhưng không có trường cha. (Cũng xem Bài tập 14.3-6.)

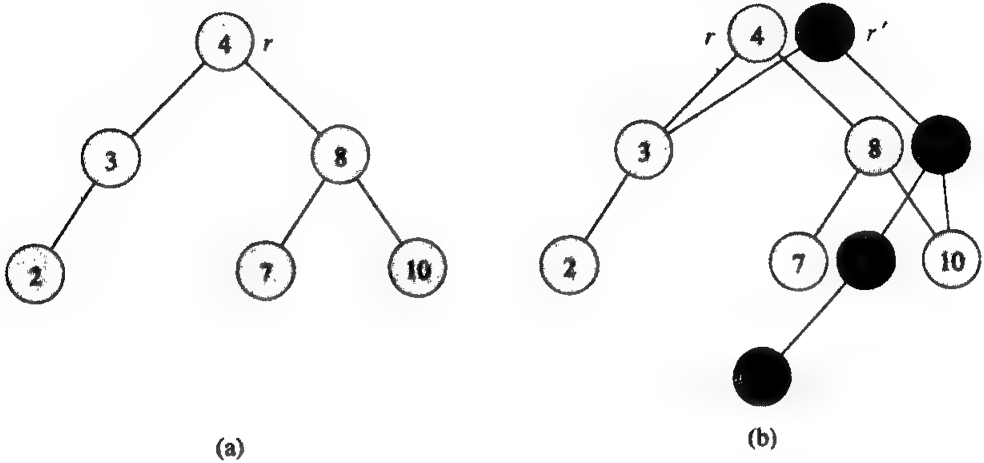
c. Nếu chiều cao của cây tìm nhị phân bền T là h, hãy nêu các yêu cầu về thời gian và không gian của việc thực thi PERSISTENT-TREE-INSERT? (Yêu cầu về không gian tỷ lệ với số lượng các nút mới được phân bổ.)

d. Giả sử ta đã gộp trường cha trong mỗi nút. Trong trường hợp này, PERSISTENT-TREE-INSERT cần tiến hành chép bổ sung. Vậy, hãy chứng minh PERSISTENT-TREE-INSERT yêu cầu $\Omega(n)$ thời gian và không gian, ở đó n là số lượng nút trong cây.

e. Nêu cách sử dụng các cây đỏ đen để bảo đảm không gian và thời gian thực hiện trường hợp xấu nhất là $O(\lg n)$ cho mỗi đợt chèn hoặc xóa.

14-2 Phép toán ghép trên các cây đỏ đen

Phép toán ***ghép*** [join] lấy hai tập hợp động S_1 và S_2 và một thành phần x sao cho với bất kỳ $x_1 \in S_1$ và $x_2 \in S_2$, ta có $key[x_1] \leq key[x] \leq key[x_2]$. Nó trả về một tập hợp $S = S_1 \cup \{x\} \cup S_2$. Trong bài toán này, ta nghiên cứu cách thực thi phép toán ghép trên các cây đỏ đen.



Hình 14.8 (a) Một cây tìm nhị phân có các khóa 2, 3, 4, 7, 8, 10. **(b)** Cây tìm nhị phân bền, kết quả của phép chèn khóa 5. Phiên bản mới nhất của tập hợp bao gồm các nút khả dụng từ gốc *r'*, và phiên bản trước đó bao gồm các nút khả dụng từ *r*. Các nút tô bóng đậm sẽ được bổ sung khi chèn khóa 5.

a. Cho một cây đỏ đen T, ta lưu trữ chiều cao đen của nó dưới dạng trường $bh[T]$. Chứng tỏ có thể duy trì trường này bằng RB-INSERT và RB-DELETE mà không yêu cầu kho lưu trữ phụ trội trong cây và không làm tăng thời gian thực hiện tiệm cận. Chứng tỏ trong khi lần xuống qua T, ta có thể xác định chiều cao đen của mỗi nút mà ta viếng thăm trong $O(1)$ thời gian cho mỗi nút đã viếng.

Ta muốn thực thi phép toán RB-JOIN(T_1, x, T_2), hủy T_1 và T_2 đồng thời trả về một cây đỏ đen $T = T_1 \cup \{x\} \cup T_2$. Cho n là tổng của các nút trong T_1 và T_2.

b. Mặc nhận mà không để mất tính tổng quát rằng $bh[T_1] \geq bh[T_2]$. Mô tả một thuật toán $O(\lg n)$-thời gian để tìm một nút đen y trong T, có khóa lớn nhất trong số các nút có chiều cao đen là $bh[T_2]$.

c. Cho T_y là cây con có gốc tại y. Mô tả cách thay T_y bằng $T_y \cup \{x\} \cup T_2$ trong $O(1)$ thời gian mà không hủy tính chất cây tìm nhị phân.

d. Nêu màu sẽ tạo cho x sao cho có thể duy trì các tính chất đỏ-đen 1, 2, và 4? Mô tả cách áp đặt tính chất 3 trong $O(\lg n)$ thời gian.

e. Chứng tỏ thời gian thực hiện của RB-JOIN là $O(\lg n)$.

Ghi chú Chương

Ý tưởng làm cân bằng một cây tìm là do Adel'son-Vel'skii và Landis [2]. Họ đã giới thiệu một lớp các cây tìm kiếm cân bằng có tên "các cây AVL" vào năm 1962. Sự cân bằng được duy trì trong các cây AVL bằng các phép quay, nhưng có thể yêu cầu tối đa $\Theta(\lg n)$ phép quay sau một phép chèn để duy trì sự cân bằng trong một cây n-nút. Một lớp cây tìm kiếm khác, có tên "các cây 2-3," đã được J. E. Hopcroft giới thiệu (không xuất bản) vào năm 1970. Sự cân bằng được duy trì trong một cây 2-3 bằng cách điều tác [manipulating] các cấp độ của các nút trong cây. Phép tổng quát hóa của các cây 2-3 được giới thiệu bởi Bayer và McCreight [18], có tên các cây B, mà ta sẽ đề cập ở Chương 19.

Các cây đỏ đen đã được Bayer [17] sáng chế dưới cái tên "các cây B nhị phân đối xứng." Guibas và Sedgewick [93] đã nghiên cứu kỹ lưỡng các tính chất của chúng và đã giới thiệu quy ước màu đỏ/đen.

Trong số nhiều biến thể khác về các cây nhị phân cân bằng, có lẽ hấp dẫn nhất là "các cây splay" của Sleator và Tarjan [177], có thể "tự điều chỉnh." (Tarjan [188] đã mô tả kỹ các cây splay.) Các cây splay duy trì sự cân bằng mà không cần bất kỳ điều kiện cân bằng tường minh nào, như màu. Thay vì thế, "các phép toán splay" (liên quan đến các phép quay) được thực hiện trong cây mỗi khi tiến hành một đợt truy cập. Hao phí khấu trừ (xem Chương 18) của mỗi phép toán trên một cây n-nút là $O(\lg n)$.

15 Tăng Cường Các Cấu Trúc Dữ Liệu

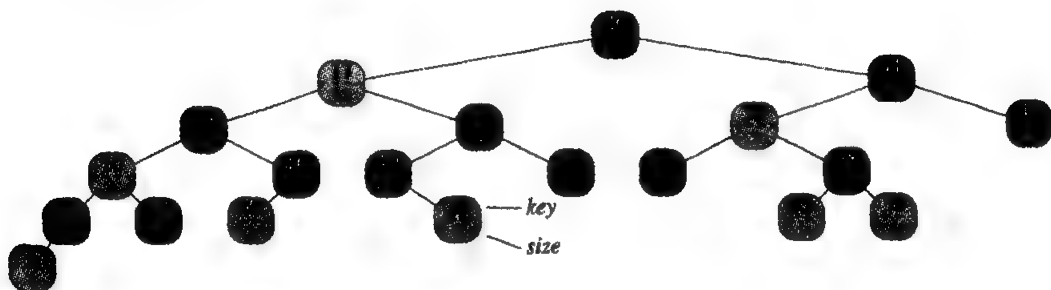
Có vài tình huống thiết kế kỹ thuật chỉ yêu cầu một cấu trúc dữ liệu "giáo khoa"—như một danh sách nối kết đôi, một bảng ánh số, hoặc một cây tìm nhị phân—song có nhiều tình huống khác yêu cầu ít nhiều óc sáng tạo. Tuy vậy, chỉ trong các tình huống hiếm hoi bạn mới cần tạo một kiểu cấu trúc dữ liệu hoàn toàn mới. Thông thường, chỉ cần tăng cường một cấu trúc dữ liệu giáo khoa bằng cách lưu trữ thông tin bổ sung trong nó. Sau đó, bạn có thể lập trình các tác vụ mới cho cấu trúc dữ liệu để hỗ trợ ứng dụng mong muốn. Tuy nhiên, việc tăng cường một cấu trúc dữ liệu không phải lúc nào cũng dễ dàng, bởi thông tin bổ sung phải được cập nhật và duy trì bởi các phép toán bình thường trên cấu trúc dữ liệu.

Chương này đề cập hai cấu trúc dữ liệu được kiến tạo bằng cách tăng cường các cây đỏ đen. Đoạn 15.1 mô tả một cấu trúc dữ liệu hỗ trợ các phép toán thống kê thứ tự chung trên một tập hợp động. Sau đó, ta có thể nhanh chóng tìm ra số nhỏ nhất thứ *i* trong một tập hợp hoặc hạng của một thành phần đã cho trong cách sắp thứ tự toàn phần của tập hợp. Đoạn 15.2 khái quát tiến trình tăng cường một cấu trúc dữ liệu và cung cấp một định lý có thể rút gọn phép tăng cường của các cây đỏ đen. Đoạn 15.3 sử dụng định lý này để giúp thiết kế một cấu trúc dữ liệu hầu duy trì một tập hợp động các quãng, chẳng hạn như các quãng thời gian. Cho một quãng truy vấn [query interval], ta có thể nhanh chóng tìm ra một quãng trong tập hợp phủ chồng nó.

15.1 Thống kê thứ tự động

Chương 10 đã giới thiệu khái niệm về một thống kê thứ tự. Cụ thể, thống kê thứ tự thứ *i* của một tập hợp *n* thành phần, ở đó $i \in \{1, 2, ..., n\}$, đơn giản là thành phần trong tập hợp có khóa nhỏ nhất thứ *i*. Ta đã thấy rằng có thể truy lục bất kỳ thống kê thứ tự nào trong $O(n)$ thời gian từ một tập hợp không sắp xếp thứ tự. Đoạn này giải thích cách sửa đổi các cây đỏ đen để có thể xác định bất kỳ thống kê thứ tự nào trong $O(\lg n)$ thời gian. Ta sẽ cũng xem cách xác định ***hạng*** [rank] của một thành

phần—tức vị trí của nó trong thứ tự tuyến tính của tập hợp—trong $O(\lg n)$ thời gian.



Hình 15.1 Một cây thống kê thứ tự, là một cây đỏ đen được tăng cường. Các nút tô bóng là đỏ, và các nút tô sẫm là đen. Ngoài các trường bình thường của nó, mỗi nút x đều có một trường $size[x]$, là số lượng nút trong cây con có gốc tại x.

Hình 15.1 có nêu một cấu trúc dữ liệu có thể hỗ trợ các phép toán thống kê thứ tự nhanh. Một ***cây thống kê thứ tự** T* đơn giản là một cây đỏ đen có thông tin bổ sung được lưu trữ trong mỗi nút. Ngoài các trường cây đỏ đen bình thường $key[x]$, $color[x]$, $p[x]$, $left[x]$, và $right[x]$ trong một nút x, ta còn có một trường khác, $size[x]$. Trường này chứa số lượng nút (trong = internal) trong cây con có gốc tại x (kể cả chính x), nghĩa là, kích cỡ của cây con. Nếu định nghĩa $size$[NIL] là 0, thì ta có đồng nhất thức

$$size[x] = size[left[x]] + size[right[x]] + 1 .$$

(Để điều quản điều kiện cận của NIL một cách đúng đắn, một thực thi thực tế có thể trắc nghiệm rõ về NIL mỗi lần truy cập trường *size* hoặc, đơn giản hơn, như trong Đoạn 14.4, dùng một cờ hiệu $nil[T]$ để biểu diễn NIL, ở đó $size[nil[T]] = 0$.)

Truy lục một thành phần có một hạng đã cho

Trước khi nêu cách duy trì thông tin kích cỡ này trong phép chèn và phép xóa, ta hãy xem xét cách thực thi hai kiểu truy vấn thống kê thứ tự dùng thông tin bổ sung này. Bắt đầu bằng một phép toán truy lục một thành phần có một hạng đã cho. Thủ tục OS-SELECT(x, i) trả về một biến trỏ đến nút chứa khóa nhỏ nhất thứ i trong cây con có gốc tại x. Để tìm khóa nhỏ nhất thứ i trong một cây thống kê thứ tự T, ta gọi OS-SELECT ($root[T]$, i).

```
OS-SELECT(x, i)
r ← size[left[x]]+1
```

```
2 if i = r
3     then return x
4 else if i < r
5     then return OS-SELECT(left[x], i)
6 else return OS-SELECT(right[x], i - r)
```

Ý tưởng đằng sau OS-SELECT cũng tương tự như trường hợp các thuật toán lựa trong Chương 10. Giá trị của *size*[*left*[*x*]] là số lượng các nút đứng trước *x* trong một tầng cây nội cấp của cây con có gốc tại *x*. Như vậy, *size*[*left*[*x*]] + 1 là hạng của *x* trong cây con có gốc tại *x*.

Trong dòng 1 của OS-SELECT, ta tính toán *r*, hạng của nút *x* trong cây con có gốc tại *x*. Nếu $i = r$, thì nút *x* là thành phần nhỏ nhất thứ *i*, do đó ta trả về *x* trong dòng 3. Nếu $i < r$, thì thành phần nhỏ nhất thứ *i* nằm trong cây con trái của *x*, do đó ta đệ quy trên *left*[*x*] trong dòng 5. Nếu $i > r$, thì thành phần nhỏ nhất thứ *i* nằm trong cây con phải *x*. Do có *r* thành phần trong cây con có gốc tại *x* đứng trước cây con phải của *x* trong một tầng cây nội cấp, thành phần nhỏ nhất thứ *i* trong cây con có gốc tại *x* là thành phần nhỏ nhất thứ $(i - r)$ trong cây con có gốc tại *right*[*x*]. Thành phần này được xác định một cách đệ quy trong dòng 6.

Để xem cách hoạt động của OS-SELECT, ta xét một đợt tìm kiếm thành phần nhỏ nhất thứ 17 trong cây thống kê thứ tự của Hình 15.1. Ta bắt đầu với *x* làm gốc, có khóa là 26, và với $i = 17$. Do kích cỡ của cây con trái của 26 là 12, hạng của nó là 13. Như vậy, ta biết nút có hạng 17 là 17 - 13 = thành phần nhỏ nhất thứ 4 trong cây con phải của 26. Sau lệnh gọi đệ quy, *x* là nút có khóa 41, và $i = 4$. Bởi kích cỡ của cây con trái của 41 là 5, hạng của nó trong cây con của nó là 6. Như vậy, ta biết nút có hạng 4 nằm trong thành phần nhỏ nhất thứ 4 trong cây con trái của 41. Sau lệnh gọi đệ quy, *x* là nút có khóa 30, và hạng của nó trong cây con của nó là 2. Như vậy, ta đệ quy một lần nữa để tìm ra 4 - 2 = thành phần nhỏ nhất thứ 2 trong cây con có gốc tại nút có khóa 38. Giờ đây, ta thấy cây con trái của nó có kích cỡ 1, có nghĩa nó là thành phần nhỏ nhất thứ hai. Như vậy, thủ tục trả về một biến trỏ đến nút có khóa 38.

Do mỗi lệnh gọi đệ quy dời xuống một cấp trong cây thống kê thứ tự, nên tổng thời gian của OS-SELECT trong ca xấu nhất là tỷ lệ với chiều cao của cây. Bởi đây là một cây đỏ đen, nên chiều cao của nó là $O(\lg n)$, ở đó *n* là số lượng nút. Như vậy, thời gian thực hiện của OS-SELECT là $O(\lg n)$ với một tập hợp động *n* thành phần.

Xác định hạng của một thành phần

Cho một biến trỏ đến một nút *x* trong một cây thống kê thứ tự *T*, thủ tục OS-RANK trả về vị trí của *x* trong thứ tự tuyến tính được xác định bởi một tầng cây nội cấp của *T*.

```
OS-RANK(T, x)
1 r ← size[left[x]] + 1
2 y ← x
3 while y ≠ root[T]
4       do if  y = right[p[y]]
5              then r ← r + size[left[p[y]]] + 1
6          y ← p[y]
7 return r
```

Thủ tục làm việc như sau. Hạng của *x* có thể được xem như số lượng nút đứng trước *x* trong một tầng cây nội cấp, cộng 1 cho chính *x*. Lượng bất biến sau đây được duy trì: tại đầu vòng lặp **while** của các dòng 3-6, *r* là hạng của *key*[*x*] trong cây con có gốc tại nút *y*. Ta duy trì lượng bất biến này như sau. Trong dòng 1, ấn định *r* là hạng của *key*[*x*] trong cây con có gốc tại *x*. Việc ấn định $y \leftarrow x$ trong dòng 2 khiến lượng bất biến là true khi lần đầu tiên thi hành đợt trắc nghiệm trong dòng 3. Trong mỗi lần lặp lại của vòng lặp **while**, ta xét cây con có gốc tại *p*[*y*]. Ta đã đếm số lượng các nút trong cây con có gốc tại nút *y* đứng trước *x* trong một tầng nội cấp [inorder walk], do đó ta phải cộng các nút trong cây con có gốc tại anh em ruột của *y* đứng trước *x* trong một tầng nội cấp, cộng 1 cho *p*[*y*] nếu nó cũng đứng trước *x*. Nếu *y* là một con trái, thì cả *p*[*y*] lẫn bất kỳ nút nào trong cây con phải của *p*[*y*] đều không đứng trước *x*, do đó ta để nguyên *r*. Bằng không, *y* là một con phải và tất cả các nút trong cây con trái của *p*[*y*] đều đứng trước *x*, như chính *p*[*y*]. Như vậy, trong dòng 5, ta cộng *size*[*left*[*y*]]+ 1 vào giá trị hiện hành của *r*. Việc ấn định $y \leftarrow p[y]$ sẽ khiến lượng bất biến là true cho lần lặp lại kế tiếp. Khi $y = root[T]$, thủ tục trả về giá trị của *r*, mà giờ đây là hạng của *key*[*x*].

Để lấy ví dụ, khi chạy OS-RANK trên cây thống kê thứ tự của Hình 15.1 để tìm hạng của nút có khóa 38, ta được dãy giá trị dưới đây của *key*[*y*] và *r* tại đầu vòng lặp **while**:

lần lặp lại	*key*[*y*]	*r*
1	38	2
2	30	4
3	41	4
4	26	17

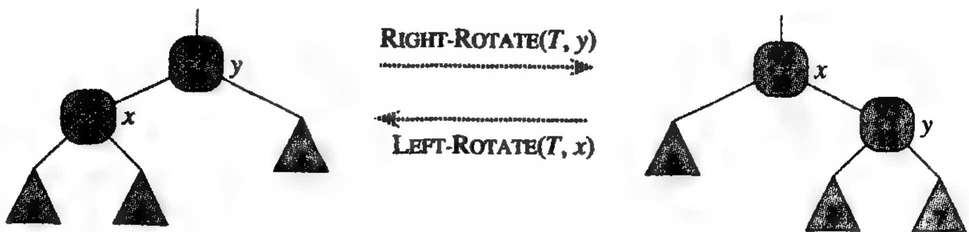
Hạng 17 được trả về.

Do mỗi lần lặp lại của vòng lặp **while** kéo dài $O(1)$ thời gian, và y dời lên một cấp trong cây với từng lần lặp lại, thời gian thực hiện của OS-RANK trong trường hợp xấu nhất sẽ tỷ lệ với chiều cao của cây: $O(\lg n)$ trên một cây thống kê thứ tự n-nút.

Duy trì kích cỡ cây con

Cho trường *size* trong mỗi nút, OS-SELECT và OS-RANK có thể nhanh chóng tính toán thông tin thống kê thứ tự. Nhưng trừ phi duy trì hiệu quả các trường này bằng các phép toán sửa đổi cơ bản trên các cây đỏ đen, công trình của chúng ta sẽ hóa ra công cốc. Giờ đây, ta chứng tỏ có thể duy trì các kích cỡ cây con cho cả phép chèn lẫn phép xóa mà không tác động đến các thời gian thực hiện tiệm cận của một trong hai phép toán.

Như đã lưu ý trong Đoạn 14.3, phép chèn vào một cây đỏ đen bao gồm hai giai đoạn. Giai đoạn đầu tiên dời xuống cây từ gốc, chèn nút mới dưới dạng một con của một nút hiện có. Giai đoạn thứ hai dời lên cây, thay đổi các màu và chung cuộc thực hiện các phép quay để duy trì các tính chất đỏ-đen.



Hình 15.2 Cập nhật các kích cỡ cây con trong các phép quay. Hai trường *size* cần được cập nhật là những trường liên thuộc trên nối kết mà phép quay được thực hiện xoay quanh nó. Các đợt cập nhật là cục bộ, chỉ yêu cầu thông tin *size* được lưu trữ trong x, y, và các gốc của các cây con được nêu dưới dạng các tam giác.

Để duy trì các kích cỡ cây con trong giai đoạn đầu tiên, ta đơn giản gia số $size[x]$ của mỗi nút x trên lộ trình băng ngang từ gốc đổ xuống về phía các lá. Nút mới bổ sung sẽ có một *size* 1. Do có $O(\lg n)$ nút trên lộ trình băng ngang, hao phí bổ sung để duy trì các trường *size* là $O(\lg n)$.

Trong giai đoạn thứ hai, các phép quay (tối đa là hai) chỉ gây ra các thay đổi về cấu trúc đối với cây đỏ đen cơ sở. Hơn nữa, phép quay là

một phép toán cục bộ: nó chỉ làm mất hiệu lực hai trường *size* trong các nút liên thuộc trên nối kết mà phép quay được thực hiện xoay quanh nó. Tham chiếu mã của LEFT-ROTATE(T,x) trong Đoạn 14.2, ta bổ sung các dòng sau:

13 $size[y] \leftarrow size[x]$

14 $size[x] \leftarrow size[left[x]] + size[right[x]] + 1$

Hình 15.2 có minh họa cách cập nhật các trường. Thay đổi đối với RIGHT-ROTATE là đối xứng.

Do có tối đa hai phép quay được thực hiện trong khi chèn vào một cây đỏ đen, nên chỉ mất thêm $O(1)$ thời gian để cập nhật các trường *size* trong giai đoạn thứ hai. Như vậy, tổng thời gian để chèn vào một cây thống kê thứ tự n-nút là $O(\lg n)$—mà theo tiệm cận cũng giống như của một cây đỏ đen bình thường.

Tiến trình xóa ra khỏi cây đỏ đen cũng bao gồm hai giai đoạn: giai đoạn đầu tiên hoạt động trên cây tìm cơ sở, và giai đoạn thứ hai tạo ra tối đa ba phép quay mà bằng không sẽ không thay đổi gì về cấu trúc. (Xem Đoạn 14.4.) Giai đoạn đầu tiên nối khử một nút y. Để cập nhật các kích cỡ cây con, ta đơn giản băng ngang một lộ trình từ nút y lên đến gốc, giảm lượng [decrementing] trường *size* của mỗi nút trên lộ trình. Do lộ trình này có chiều dài $O(\lg n)$ trong một cây đỏ đen n-nút, nên thời gian bổ sung bỏ ra để duy trì các trường *size* trong giai đoạn đầu tiên là $O(\lg n)$. $O(1)$ phép quay trong giai đoạn thứ hai của tiến trình xóa có thể được điều quản giống như trường hợp chèn. Như vậy, cả hai phép chèn và xóa, kể cả việc duy trì các trường *size*, chiếm $O(\lg n)$ thời gian cho một cây thống kê thứ tự n-nút.

Bài tập

15.1-1

Nêu cách hoạt động của OS-SELECT(T, 10) trên cây đỏ đen T của Hình 15.2.

15.1-2

Nêu cách hoạt động của OS-RANK (T, x) trên cây đỏ đen T của Hình 15.2 và nút x có $key[x] = 35$.

15.1-3

Viết một phiên bản không đệ quy của OS-SELECT.

15.1-4

Viết một thủ tục đệ quy OS-KEY-RANK(T,k) lấy một cây thống kê

thứ tự T và một khóa k làm nhập liệu và trả về hạng của k trong tập hợp động được biểu diễn bởi T. Giả sử các khóa của T là riêng biệt.

15.1-5

Cho một thành phần x trong một cây thống kê thứ tự n-nút và một số tự nhiên i, làm sao để có thể xác định phần tử kế vị thứ i của x trong thứ tự tuyến tính của cây được xác định trong $O(\lg n)$ thời gian?

15.1-6

Nhận thấy mỗi khi trường *size* được tham chiếu trong OS-SELECT hoặc OS-RANK, nó chỉ được dùng để tính toán hạng của x trong cây con có gốc tại x. Do đó, giả sử trong mỗi nút, ta lưu trữ hạng của nó trong cây con mà nó là gốc. Nêu cách duy trì thông tin này trong phép chèn và phép xóa. (Nên nhớ, hai phép toán này có thể tạo ra các phép quay.)

15.1-7

Nêu cách sử dụng một cây thống kê thứ tự để đếm số lượng các phép nghịch đảo (xem Bài toán 1-3) trong một mảng có kích cỡ n trong thời gian $O(n \lg n)$.

15.1-8

Xét n dây cung trên một vòng tròn, mỗi dây được định nghĩa bởi các điểm cuối của nó. Mô tả một thuật toán $O(n \lg n)$ thời gian để xác định số lượng các cặp dây cung giao nhau bên trong vòng tròn. (Ví dụ, nếu n dây cung là tất cả các đường kính gặp tại tâm, thì đáp án đúng đắn là $\binom{n}{2}$. Giả sử không có hai dây cung chia sẻ một điểm cuối.

15.2 Cách tăng cường một cấu trúc dữ liệu

Tiến trình tăng cường một cấu trúc dữ liệu cơ bản để hỗ trợ công năng bổ sung xảy ra khá phổ biến trong thiết kế thuật toán. Nó sẽ được dùng lại trong đoạn sau để thiết kế một cấu trúc dữ liệu hỗ trợ các phép toán trên các quãng. Trong đoạn này, ta xem xét các bước liên quan đến phép tăng cường đó. Ta cũng sẽ chứng minh một định lý cho phép dễ dàng tăng cường các cây đỏ đen trong nhiều trường hợp. Tiến trình tăng cường một cấu trúc dữ liệu có thể được tách nhỏ thành bốn bước:

1. chọn một cấu trúc dữ liệu cơ sở,

2. xác định thông tin bổ sung sẽ được duy trì trong cấu trúc dữ liệu cơ sở,

3. xác minh thông tin bổ sung có thể được duy trì cho các phép toán

sửa đổi căn bản trên cấu trúc dữ liệu cơ sở, và

4. phát triển các phép toán mới.

Giống như mọi phương pháp thiết kế theo quy tắc [prescriptive], bạn không nên mù quáng theo các bước theo thứ tự đã cho. Phần lớn công trình thiết kế đều chứa một thành phần thử và lỗi [trial and error], và tiến độ trên tất cả các bước thường tiến hành song song. Ví dụ, không có điểm nào trong việc xác định thông tin bổ sung và phát triển các phép toán mới (các bước 2 và 4) nếu ta không thể duy trì thông tin bổ sung một cách hiệu quả. Tuy vậy, phương pháp bốn bước này sẽ giúp bạn tập trung nỗ lực tốt hơn trong việc tăng cường một cấu trúc dữ liệu, và cũng là một cách tốt để tổ chức sưu liệu một dữ liệu cấu trúc đã tăng cường.

Ta đã theo các bước này trong Đoạn 15.1 để thiết kế các cây thống kê thứ tự. Với bước 1, ta đã chọn các cây đỏ đen làm cấu trúc dữ liệu cơ sở. Tính thích hợp của các cây đỏ đen được thể hiện qua sự hỗ trợ hiệu quả của các phép toán tập hợp động khác trên một thứ tự tổng, như MINIMUM, MAXIMUM, SUCCESSOR, và PREDECESSOR.

Với bước 2, ta đã cung cấp các trường *size*, mà trong mỗi nút x lưu trữ kích cỡ của cây con có gốc tại x. Nói chung, thông tin bổ sung khiến các phép toán trở nên hiệu quả hơn. Ví dụ, ta đã có thể dùng chính các khóa được lưu trữ trong cây để thực thi OS-SELECT và OS-RANK, nhưng chúng sẽ không chạy trong $O(\lg n)$ thời gian. Đôi lúc, thông tin bổ sung là thông tin biến trỏ thay vì dữ liệu, như trong Bài tập 15.2-1.

Với bước 3, ta đã bảo đảm phép chèn và phép xóa có thể duy trì các trường *size* trong khi vẫn chạy trong $O(\lg n)$ thời gian. Về mặt lý tưởng, một lượng thay đổi nhỏ đối với cấu trúc dữ liệu cũng đủ để duy trì thông tin bổ sung. Ví dụ, nếu trong mỗi nút ta đơn giản lưu trữ hạng của nó trong cây, các thủ tục OS-SELECT và OS-RANK sẽ chạy nhanh, song việc chèn một thành phần cực tiểu mới sẽ tạo ra một thay đổi đối với thông tin này trong mọi nút của cây. Thay vì thế, khi ta lưu trữ các kích cỡ cây con, việc chèn một thành phần mới sẽ chỉ khiến thông tin thay đổi trong $O(\lg n)$ nút mà thôi.

Với bước 4, ta đã phát triển các phép toán OS-SELECT và OS-RANK. Xét cho cùng, nhu cầu về các phép toán mới chính là lý do tại sao ta lại quan tâm hàng đầu đến việc tăng cường một dữ liệu cấu trúc. Thỉnh thoảng, thay vì phát triển các phép toán mới, ta dùng thông tin bổ sung để giải quyết các phép toán hiện có, như trong Bài tập 15.2-1.

Tăng cường các cây đỏ đen

Khi các cây đỏ đen làm cơ sở cho một cấu trúc dữ liệu đã tăng

cường, ta có thể chứng minh một số kiểu thông tin bổ sung có thể luôn được duy trì hiệu quả bởi phép chèn và phép xóa, nhờ đó khiến cho bước 3 trở nên rất dễ dàng. Phần chứng minh định lý dưới đây cũng tương tự như đối số trong Đoạn 15.1 rằng trường *size* có thể được duy trì cho các cây thống kê thứ tự.

Định lý 15.1 (Tăng cường một cây đỏ đen)

Cho *f* là một trường tăng cường một cây đỏ đen *T* gồm *n* nút, và giả sử nội dung của *f* cho một nút *x* có thể được tính toán chỉ dùng thông tin trong các nút *x*, *left*[*x*], và *right*[*x*], kể cả *f*[*left*[*x*]] và *f*[*right*[*x*]]. Thì, ta có thể duy trì các giá trị của *f* trong tất cả các nút của *T* trong khi chèn và xóa mà không tác động theo tiệm cận đến khả năng thực hiện $O(\lg n)$ của các phép toán này.

Chứng minh Ý tưởng chính của phần chứng minh đó là một thay đổi đối với một trường *f* trong một nút *x* sẽ chỉ lan truyền đến các tiền bối của *x* trong cây. Nghĩa là, việc thay đổi *f*[*x*] có thể yêu cầu cập nhật *f*[*p*[*x*]], mà không còn gì khác; việc cập nhật *f*[*p*[*x*]] có thể yêu cầu cập nhật *f*[*p*[*p*[*x*]]], mà không còn gì khác; cứ thế tiếp tục lên cây. Khi *f*[*root*[*T*]] được cập nhật, thì không còn nút nào khác phụ thuộc vào giá trị mới, do đó tiến trình kết thúc. Do chiều cao của một cây đỏ đen là $O(\lg n)$, nên việc thay đổi của trường *f* trong một nút sẽ hao phí $O(\lg n)$ thời gian để cập nhật các nút lệ thuộc vào sự thay đổi đó.

Tiến trình chèn một nút *x* vào *T* bao gồm hai giai đoạn. (Xem Đoạn 14.3.) Trong giai đoạn đầu, *x* được chèn dưới dạng một con của một nút *p*[*x*] hiện có. Giá trị của *f* [*x*] có thể được tính toán trong $O(1)$ thời gian bởi, theo giả thiết, nó chỉ tùy thuộc vào thông tin trong các trường khác của chính *x* và thông tin trong các con của *x*, mà các con của *x* lại là NIL. Sau khi tính toán *f* [*x*], sự thay đổi lan truyền đi lên cây. Như vậy, tổng thời gian cho giai đoạn đầu của phép chèn là $O(\lg n)$. Trong giai đoạn thứ hai, các phép quay chỉ thay đổi cấu trúc của cây. Do chỉ có hai nút thay đổi trong một phép quay, nên tổng thời gian để cập nhật các trường *f* là $O(\lg n)$ với mỗi phép quay. Do số lượng phép quay trong khi chèn tối đa là hai, nên tổng thời gian cho phép chèn là $O(\lg n)$.

Giống như phép chèn, tiến trình xóa cũng có hai giai đoạn. (Xem Đoạn 14.4.) Trong giai đoạn đầu, cây sẽ thay đổi nếu nút bị xóa được thay bằng phần tử kế vị của nó, và rồi cũng vậy khi nối khử [splice out] nút bị xóa hoặc phần tử kế vị của nó. Việc lan truyền các chi tiết cập nhật đối với *f* do các thay đổi này gây ra sẽ hao phí tối đa $O(\lg n)$ bởi các thay đổi sửa đổi cây theo cục bộ. Tiến trình chỉnh sửa cây đỏ đen trong giai đoạn thứ hai yêu cầu tối đa ba phép quay, và mỗi phép quay yêu cầu tối đa $O(\lg n)$ thời gian để lan truyền các chi tiết cập nhật đối với *f*.

Do đó, cũng như phép chèn, tổng thời gian để xóa là $O(\lg n)$.

Trong nhiều trường hợp, chẳng hạn như việc duy trì các trường *size* trong các cây thống kê thứ tự, mức hao phí cập nhật sau một phép quay là $O(1)$, thay vì $O(\lg n)$ được dẫn xuất trong phần chứng minh của Định lý 15.1. Bài tập 15.2-4 cho ta một ví dụ.

Bài tập

15.2-1

Nêu cách hỗ trợ từng đợt truy vấn tập hợp động MINIMUM, MAXIMUM, SUCCESSOR, và PREDECESSOR trong $O(1)$ thời gian ca xấu nhất trên một cây thống kê thứ tự đã tăng cường. Không được tác động đến khả năng thực hiện tiệm cận của các phép toán khác trên các cây thống kê thứ tự.

15.2-2

Có thể duy trì các chiều cao đen của các nút trong một cây đỏ đen dưới dạng các trường trong các nút của cây mà không tác động đến khả năng thực hiện tiệm cận của các phép toán trên cây đỏ đen không? Nêu cách thực hiện, hoặc chứng tỏ tại sao không.

15.2-3

Có thể duy trì hiệu quả các chiều sâu của các nút trong một cây đỏ đen dưới dạng các trường trong các nút của cây hay khong? Nêu cách thực hiện, hoặc chứng tỏ tại sao không.

15.2-4 *

Cho $\otimes$ là toán tử nhị phân kết hợp, và cho a là một trường được duy trì trong mỗi nút của một cây đỏ đen. Giả sử trong mỗi nút x, ta muốn gộp thêm một trường f sao cho $f[x] = a[x_1] \otimes a[x_2] \otimes \dots \otimes a[x_m]$, ở đó x_1, $x_2, \dots, x_m$, là bảng kê nội cấp của các nút trong cây con có gốc tại x. Chứng tỏ các trường f có thể cập nhật đúng đắn trong $O(1)$ thời gian sau một phép quay. Sửa đổi đối số của bạn chút đỉnh để chứng tỏ có thể duy trì các trường *size* trong các cây thống kê thứ tự trong $O(1)$ thời gian cho mỗi phép quay.

15.2-5 *

Ta muốn tăng cường các cây đỏ đen bằng một phép toán RB-ENUMERATE(x, a, b) kết xuất tất cả các khóa k sao cho $a \leq k \leq b$ trong một cây đỏ đen có gốc tại x. Mô tả cách thực thi RB-ENUMERATE trong $\Theta(m + \lg n)$ thời gian, ở đó m là số lượng các khóa kết xuất và n là số lượng nút trong [internal nodes] trong cây. (*Mách nước:* Không cần bổ sung các trường mới vào cây đỏ đen.)

15.3 Các cây quãng

Trong đoạn này, ta sẽ tăng cường các cây đỏ đen để hỗ trợ các phép toán trên các tập hợp động gồm các quãng. Một ***quãng đóng*** [closed interval] là một cặp số thực $[t_1, t_2]$ có sắp xếp thứ tự, với $t_1 \leq t_2$. Quãng $[t_1, t_2]$ biểu diễn tập hợp $\{t \in \mathbf{R} : t_1 \leq t \leq t_2\}$. Các ***quãng mở*** và ***nửa mở*** bỏ qua cả hai hoặc một trong các điểm cuối từ tập hợp, theo thứ tự nêu trên. Trong đoạn này, ta mặc nhận các quãng ở tình trạng đóng; về khái niệm, tiến trình mở rộng các kết quả ra các quãng mở và nửa mở.

Các quãng là phương cách tiện dụng để biểu thị các sự kiện mà mỗi sự kiện choán một quãng thời gian liên tục. Ví dụ, ta muốn truy vấn một cơ sở dữ liệu có các quãng thời gian để phát hiện những sự kiện nào đã xảy ra trong một quãng đã cho. Cấu trúc dữ liệu trong đoạn này cung cấp một biện pháp hiệu quả để duy trì một cơ sở dữ liệu quãng như vậy.

Ta có thể biểu diễn một quãng $[t_1, t_2]$ dưới dạng một đối tượng i, có các trường $low[i] = t_1$ (***điểm cuối thấp***) và $high[i] = t_2$ (***điểm cuối cao***). Ta nói rằng các quãng i và i' ***phủ chồng*** nếu $i \cap i' \neq \emptyset$, nghĩa là, nếu $low[i] \leq high[i']$ và $low[i'] \leq high[i]$. Bất kỳ hai quãng i và i' nào đều thỏa ***phép tam phân quãng*** [interval trichotomy]; nghĩa là, chính xác một trong ba tính chất dưới đây tồn tại:

a. i và i' phủ chồng,

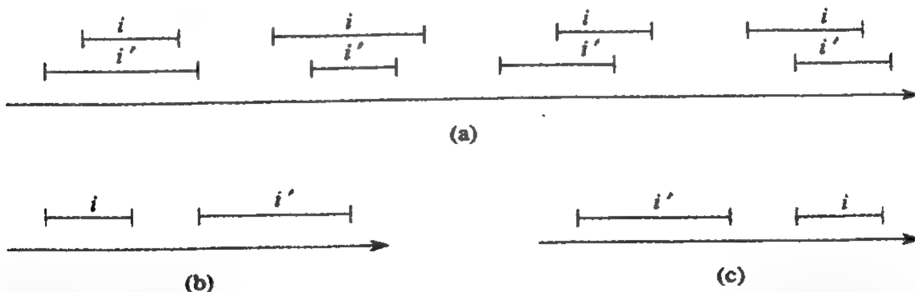
b. $high[i] < low[i']$,

c. $high[i'] < low[i]$.

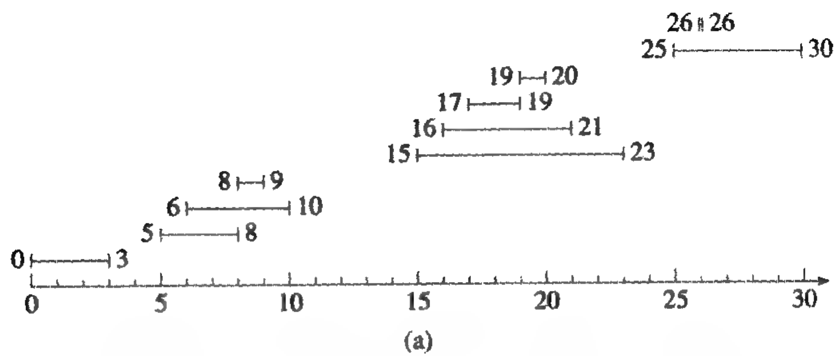
Hình 15.3 có nêu ba khả năng.

Một ***cây quãng*** [interval tree] là một cây đỏ đen duy trì một tập hợp động các thành phần, với mỗi thành phần x chứa một quãng $int[x]$. Các cây quãng hỗ trợ các phép toán sau.

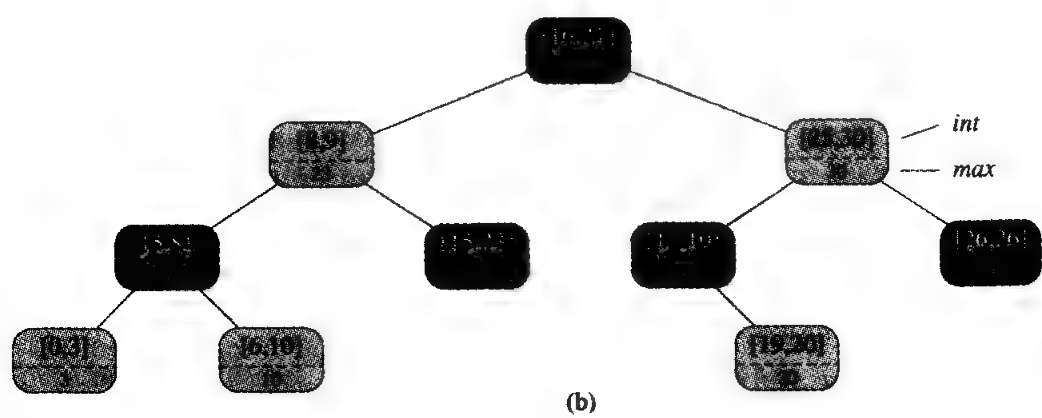
INTERVAL-INSERT(T,x) bổ sung thành phần x, có trường int được mặc nhận chứa một quãng, vào cây quãng T.



Hình 15.3 Phép tam phân quãng của hai quãng đóng i và i'. **(a)** Nếu i và i' phủ chồng, ta có bốn tình huống; trong mỗi tình huống, $low[i] \leq high[i']$ và $low[i'] \leq high[i]$. **(b)** $high[i] < low[i']$. **(c)** $high[i'] < low[i]$.



(a)



(b)

Hình 15.4 Một cây quãng. **(a)** Một tập hợp 10 quãng, được nêu ở dạng sắp xếp thứ tự từ dưới lên theo điểm cuối trái. **(b)** Cây quãng biểu diễn chúng. Một tầng cây nội cấp của cây liệt kê các nút được sắp xếp thứ tự theo điểm cuối trái.

INTERVAL-DELETE(T,x) gỡ bỏ thành phần x ra khỏi cây quãng T.

INTERVAL-SEARCH(T, i) trả về một biến trỏ đến đến thành phần x trong cây quãng T sao cho $int[x]$ phủ chồng quãng i, hoặc NIL nếu không có thành phần nào như vậy nằm trong tập hợp.

Hình 15.4 nêu cách thức mà một cây quãng biểu diễn một tập hợp các quãng. Ta sẽ lần theo phương pháp bốn-bước trong Đoạn 15.2 khi ta xem lại thiết kế của một cây quãng và các phép toán chạy trên nó.

Bước 1: Cấu trúc dữ liệu cơ sở

Ta chọn một cây đỏ đen ở đó mỗi nút x chứa một quãng $int[x]$ và khóa của x là điểm cuối thấp, $low[int[x]]$, của quãng. Như vậy, một tầng cây nội cấp của cấu trúc dữ liệu liệt kê các quãng được sắp xếp thứ tự theo điểm cuối thấp.

Bước 2: Thông tin bổ sung

Ngoài chính các quãng, mỗi nút *x* chứa một giá trị *max*[*x*], là giá trị cực đại của bất kỳ điểm cuối quãng nào được lưu trữ trong cây con có gốc tại *x*. Do điểm cuối cao của một quãng bất kỳ ít nhất cũng lớn bằng điểm cuối thấp của nó, nên *max*[*x*] là giá trị cực đại của tất cả các điểm cuối phải trong cây con có gốc tại *x*.

Bước 3: Duy trì thông tin

Ta phải xác minh có thể thực hiện phép chèn và phép xóa trong $O(\lg n)$ thời gian trên một cây quãng gồm *n* nút. Ta có thể xác định *max*[*x*] căn cứ vào quãng *int*[*x*] và các giá trị *max* của các con của nút *x*:

$$max[x] = \max(high[int[x]], max[left[x]], max[right[x]]) .$$

Như vậy, theo Định lý 15.1, phép chèn và phép xóa chạy trong $O(\lg n)$ thời gian. Thực tế, có thể hoàn tất tiến trình cập nhật các trường *max* sau một phép quay trong $O(1)$ thời gian, như đã nêu trong Bài tập 15.2-4 và 15.3-1.

Bước 4: Phát triển các phép toán mới

Phép toán mới duy nhất mà ta cần đó là INTERVAL-SEARCH(*T*, *i*), nó tìm một quãng trong cây *T* phủ chồng quãng *i*. Nếu không có quãng nào phủ chồng *i* trong cây, nó trả về NIL.

```
INTERVAL-SEARCH(T, i)
x ← root[T]
while x ≠ NIL và i không phủ chồng int[x]
    do if left[x] ≠ NIL và max[left[x]] ≥ low[i]
          then x ← left[x]
          else x ← right[x]
return x
```

Đợt tìm kiếm một quãng phủ chồng *i* bắt đầu với *x* tại gốc của cây và tiếp tục đi xuống. Nó kết thúc khi tìm thấy một quãng phủ chồng hoặc *x* trở thành NIL. Bởi mỗi lần lặp lại của vòng lặp căn bản kéo dài $O(1)$ thời gian, và bởi chiều cao của một cây đỏ đen *n*-nút là $O(\lg n)$, nên thủ tục INTERVAL-SEARCH mất $O(\lg n)$ thời gian.

Trước khi tìm hiểu tại sao INTERVAL-SEARCH lại đúng, ta hãy xem xét cách làm việc của nó trên cây quãng trong Hình 15.4. Giả sử ta muốn tìm một quãng phủ chồng quãng $i = [22, 25]$. Ta bắt đầu với *x* làm gốc, nó chứa [16,21] và không phủ chồng *i*. Do $max[left[x]] = 23$ lớn hơn $low[i] = 22$, vòng lặp tiếp tục với *x* làm con trái của gốc—nút chứa [8, 9], cũng không phủ chồng *i*. Lần này, $max[left[x]] = 10$ nhỏ hơn

low[*i*] = 22, do đó vòng lặp tiếp tục với con phải của *x* làm *x* mới. Quãng [15, 23] được lưu trữ trong nút này sẽ phủ chồng *i*, do đó thủ tục trả về nút này.

Để lấy một ví dụ về đợt tìm kiếm không thành công, giả sử ta muốn tìm một quãng phủ chồng *i* = [11, 14] trong cây quãng của Hình 15.4. Một lần nữa ta lại bắt đầu với *x* làm gốc. Bởi quãng [16, 21] của gốc không phủ chồng *i*, và bởi *max*[*left*[*x*]] = 23 lớn hơn *low*[*i*] = 11, ta quay sang trái đến nút chứa [8, 9]. (Lưu ý, không có quãng nào trong cây con phải phủ chồng *i*—ta sẽ nêu lý do tại sao về sau.) Quãng [8, 9] không phủ chồng *i*, và *max*[*left*[*x*]] = 10 nhỏ hơn *low*[*i*] = 11, do đó ta quay sang phải. (Lưu ý, không có quãng nào trong cây con trái phủ chồng *i*.) Quãng [15,23] không phủ chồng *i*, và con trái của nó là NIL, do đó ta quay sang phải, vòng lặp kết thúc, và NIL được trả về.

Để xem tại sao INTERVAL-SEARCH lại đúng, ta phải hiểu tại sao chỉ cần xem xét một lộ trình đơn lẻ từ gốc. Ý tưởng cơ bản đó là tại một nút *x* bất kỳ, nếu *int*[*x*] không phủ chồng *i*, đợt tìm kiếm luôn tiến hành theo một hướng an toàn: một quãng phủ chồng sẽ dứt khoát được tìm thấy nếu như có nó trong cây. Định lý dưới đây phát biểu tính chất này một cách chính xác hơn.

Định lý 15.2

Xét bất kỳ lần lặp lại nào của vòng lặp **while** trong khi thi hành INTERVALSEARCH(*T*, *i*).

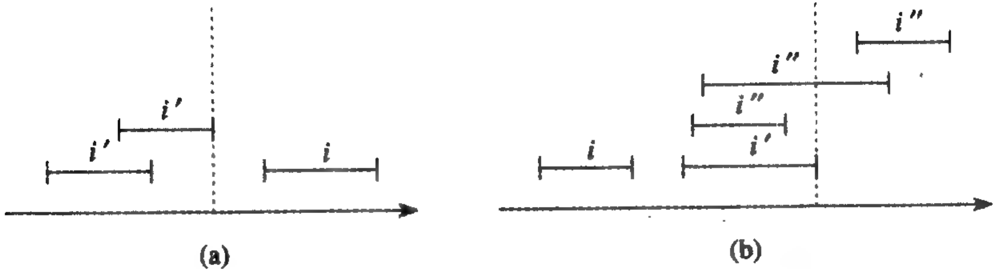
1. Nếu dòng 4 được thi hành (đợt tìm kiếm quay sang trái), thì cây con trái của *x* chứa một quãng phủ chồng *i* hoặc không có quãng nào trong cây con phải của *x* phủ chồng *i*.

2. Nếu dòng 5 được thi hành (đợt tìm kiếm quay sang phải), thì cây con trái của *x* không chứa quãng nào phủ chồng *i*.

Chứng minh Phần chứng minh của cả hai trường hợp tùy thuộc vào phép tam phân quãng. Ta chứng minh trường hợp 2 trước, bởi nó đơn giản hơn. Nhận thấy nếu dòng 5 được thi hành, thì do điều kiện rẽ nhánh trong dòng 3, ta có *left*[*x*] = NIL, hoặc *max*[*left*[*x*]] < *low*[*i*]. Nếu *left*[*x*] = NIL, cây con có gốc tại *left*[*x*] rõ ràng không chứa quãng nào phủ chồng *i*, bởi vì nó không chứa quãng nào cả. Do đó giả sử, *left*[*x*] ≠ NIL và *max*[*left*[*x*]] < *low*[*i*]. Cho *i'* là một quãng trong cây con trái của *x*. (Xem Hình 15.5(**a**).) Do *max*[*left*[*x*]] là điểm cuối lớn nhất trong cây con trái của *x*, nên ta có

$$\begin{aligned} high[i'] &\leq max[left[x]] \\ &< low[i], \end{aligned}$$

và như vậy, theo phép tam phân quãng, *i'* và *i* không phủ chồng, hoàn tất phần chứng minh của trường hợp 2.



Hình 15.5 Các quãng trong phần chứng minh của Định lý 15.2. Giá trị *max*[*left*[*x*]] được nêu trong mỗi trường hợp dưới dạng một vạch chấm cách. **(a)** Trường hợp 2: đợt tìm kiếm quay sang phải. Không có quãng *i'* nào có thể phủ chồng *i*. **(b)** Trường hợp 1: đợt tìm kiếm quay sang trái. Cây con trái của *x* chứa một quãng phủ chồng *i* (tình huống không được nêu), hoặc có một quãng *i'* trong cây con trái của *x* sao cho *high*[*i'*] = *max*[*left*[*x*]]. Do *i* không phủ chồng *i'*, nên nó cũng không phủ chồng bất kỳ quãng *i"* nào trong cây con phải của *x*, bởi *low*[*i'*] ≤ *low*[*i"*].

Để chứng minh trường hợp 1, ta có thể mặc nhận rằng không có quãng nào trong cây con trái của *x* phủ chồng *i* (bởi nếu có, ta coi như đã xong), và như vậy ta chỉ cần chứng minh không có quãng nào trong cây con phải của *x* phủ chồng *i* theo giả thiết này. Nhận thấy nếu dòng 4 được thi hành, thì do điều kiện rẽ nhánh trong dòng 3, ta có *max*[*left*[*x*]] ≥ *low*[*i*]. Hơn nữa, theo định nghĩa của trường *max*, ta phải có một quãng *i'* nào đó trong cây con trái của *x* sao cho

$$\begin{aligned} high[i'] &= max[left[x]] \\ &\geq low[i]. \end{aligned}$$

(Hình 15.5(b) minh họa tình huống này.) Bởi *i* và *i'* không phủ chồng, và bởi không đúng rằng *high*[*i'*] < *low*[*i*], nên nó theo phép tam phân quãng *high*[*i*] < *low*[*i'*]. Các cây quãng có khóa trên các điểm cuối thấp của các quãng, và như vậy tính chất cây tìm kiếm hàm ý rằng với bất kỳ quãng *i"* nào trong cây con phải của *x*,

$$\begin{aligned} high[i] &< low[i'] \\ &\leq low[i'']. \end{aligned}$$

Theo phép tam phân quãng, *i* và *i"* không phủ chồng.

Định lý 15.2 bảo đảm nếu INTERVAL-SEARCH tiếp tục với một trong các con của *x* và không tìm thấy quãng phủ chồng nào, một đợt tìm kiếm bắt đầu với con khác của *x* sẽ hóa ra công cốc.

Bài tập

15.3-1

Viết mã giả cho LEFT-ROTATE hoạt động trên các nút trong một cây quãng và cập nhật các trường *max* trong $O(1)$ thời gian.

15.3-2

Viết lại mã của INTERVAL-SEARCH để nó làm việc đúng đắn khi tất cả các quãng được mặc nhận là mở.

15.3-3

Mô tả một thuật toán hiệu quả mà, căn cứ vào một quãng i, trả về một quãng phủ chồng i có điểm cuối thấp cực tiểu, hoặc NIL nếu không có quãng nào như vậy tồn tại.

15.3-4

Cho một cây quãng T và một quãng i, mô tả cách liệt kê tất cả các quãng trong T phủ chồng i trong $O(\min(n, k \lg n))$ thời gian, ở đó k là số lượng các quãng trong danh sách kết xuất. (Tùy chọn: Tìm một giải pháp không sửa đổi cây.)

15.3-5

Gợi ý các sửa đổi cho các thủ tục cây quãng để hỗ trợ phép toán INTERVAL-SEARCH-EXACTLY(T, i), trả về một biến trỏ đến một nút x trong cây quãng T sao cho $low[int[x]] = low[i]$ và $high[int[x]] = high[i]$, hoặc NIL nếu T không chứa nút nào như vậy. Tất cả các phép toán, kể cả INTERVALSEARCH-EXACTLY, sẽ chạy trong $O(\lg n)$ thời gian trên một cây n-nút.

15.3-6

Nêu cách duy trì một tập hợp động Q gồm các con số hỗ trợ phép toán MIN-GAP, cho ra tầm lớn của hiệu của hai số gần nhất trong Q. Ví dụ, nếu $Q = \{1, 5, 9, 15, 18, 22\}$, thì MIN-GAP($Q$) trả về 18 - 15 = 3, bởi 15 và 18 là hai số gần nhất trong Q. Tạo các phép toán INSERT, DELETE, SEARCH, và MINGAP càng hiệu quả càng tốt, và phân tích thời gian thực hiện của chúng.

15.3-7 *

Các cơ sở dữ liệu VLSI thường biểu diễn một mạch tích hợp dưới dạng một danh sách các hình chữ nhật. Giả sử mỗi hình chữ nhật được hướng thẳng (các cạnh song song với trục x và y), sao cho phần biểu diễn của một hình chữ nhật bao gồm các tọa độ x và y cực tiểu và cực

đại của nó. Nêu một thuật toán $O(n \lg n)$-thời gian để quyết định xem một tập hợp các hình chữ nhật được biểu diễn như vậy có chứa hai hình chữ nhật phủ chồng hay không. Thuật toán của bạn không cần báo cáo tất cả các cặp giao nhau, nhưng nó phải báo cáo tồn tại một khả năng phủ chồng nếu một hình chữ nhật phủ nguyên cả một hình chữ nhật khác, cho dù các đường biên không giao nhau. (*Mách nước*: Dời một đường "quét" [sweep line] qua tập hợp các hình chữ nhật.)

Bài toán

15-1 Điểm phủ chồng cực đại

Giả sử ta muốn theo dõi một ***điểm phủ chồng cực đại*** [point of maximum overlap] trong một tập hợp các quãng—một điểm có số quãng lớn nhất trong cơ sở dữ liệu phủ chồng nó. Nêu cách duy trì hiệu quả điểm phủ chồng cực đại trong khi các quãng được chèn và xóa.

15-2 Phép hoán vị Josephus

Bài toán Josephus được định nghĩa như sau. Giả sử n người được bố trí trong một vòng tròn và ta có một số nguyên dương $m \leq n$. Bắt đầu bằng một người đầu tiên được chỉ định, ta tiếp tục vòng quanh vòng tròn, gỡ bỏ mọi người thứ m. Sau khi gỡ bỏ từng người, tiến trình đếm tiếp tục quanh vòng tròn còn lại. Tiến trình tiếp tục cho đến khi tất cả n người được gỡ bỏ. Thứ tự mà mọi người được gỡ bỏ ra khỏi vòng tròn sẽ định nghĩa ***phép hoán vị Josephus*** (n, m) của các số nguyên $1, 2, \ldots, n$. Ví dụ, phép hoán vị Josephus (7, 3) là $\langle 3, 6, 2, 7, 5, 1, 4 \rangle$.

a. Giả sử m là một hằng. Mô tả một thuật toán $O(n)$-thời gian mà, căn cứ vào một số nguyên n, kết xuất phép hoán vị Josephus (n, m).

b. Giả sử m không phải là một hằng. Mô tả một thuật toán $O(n \lg n)$-thời gian mà, căn cứ vào các số nguyên n và m, kết xuất phép hoán vị Josephus (n, m).

Ghi chú Chương

Preparata và Shamos [160] mô tả một số cây quãng xuất hiện trong tài liệu. Trong số các cây quan trọng hơn về lý thuyết ta có các cây được coi là của riêng H. Edelsbrunner (1980) và E. M. McCreight (1981), mà, trong một cơ sở dữ liệu n quãng, cho phép điểm danh tất cả k quãng phủ chồng một quãng truy vấn đã cho trong $O(k + \lg n)$ thời gian.

IV *Các Kỹ Thuật Phân Tích và Thiết Kế Cao Cấp*

Mở đầu

Phần này đề cập ba kỹ thuật quan trọng để thiết kế và phân tích các thuật toán hiệu quả: lập trình động (Chương 16), các thuật toán tham (Chương 17), và phân tích khấu trừ (Chương 18). Các phần trên đây đã trình bày các kỹ thuật có thể áp dụng rộng rãi khác, như chia để trị, ngẫu nhiên hóa, và giải pháp của các phép truy toán. Các kỹ thuật mới có hơi phức tạp hơn, nhưng chúng cần thiết để giải quyết hiệu quả nhiều bài toán điện toán. Các chủ đề giới thiệu trong phần này sẽ được lặp lại về sau trong sách này.

Lập trình động [dynamic programming] thường áp dụng cho các bài toán tối ưu hóa ở đó một tập hợp các chọn lựa phải được thực hiện để đạt đến một giải pháp tối ưu. Khi thực hiện các chọn lựa, các bài toán con cùng dạng tương tự thường nảy sinh. Lập trình động tỏ ra hiệu quả khi một bài toán con đã cho có thể nảy sinh từ nhiều tập hợp bộ phận các chọn lựa; kỹ thuật chính đó là lưu trữ, hoặc "memo hóa"[memoize], giải pháp của từng bài toán con như vậy để phòng trường hợp nó tái xuất hiện. Chương 16 nêu cách thức mà ý tưởng đơn giản này có thể dễ dàng biến đổi các thuật toán thời gian mũ [exponential time] thành các thuật toán thời gian đa thức [polynomial-time].

Giống như các thuật toán lập trình động, các thuật toán tham [greedy algorithm] cũng thường áp dụng cho các bài toán tối ưu hóa ở đó một tập hợp các chọn lựa phải được thực hiện để đạt đến một giải pháp tối ưu. Ý tưởng của thuật toán tham đó là thực hiện từng chọn lựa theo cách tối ưu cục bộ. Một ví dụ đơn giản đó là việc đổi đồng xu: để giảm thiểu số lượng đồng xu cần thay đổi theo một lượng đã cho, ta chỉ cần lựa lặp đi lặp lại đồng xu loại lớn nhất không lớn hơn lượng vẫn nợ. Có nhiều bài toán như vậy ở đó cách tiếp cận tham cung cấp một giải pháp tối ưu nhanh hơn nhiều so với cách tiếp cận lập trình động. Tuy nhiên, không phải lúc nào cũng có thể dễ dàng xác định cách tiếp cận tham sẽ hiệu quả hay không. Chương 17 ôn lại lý thuyết tạng ma trận [matroid], thường hữu ích đối với kiểu xác định như vậy.

Phân tích khấu trừ [amortized analysis] là một công cụ để phân tích các thuật toán thực hiện một dãy tương tự các phép toán. Thay vì định cận mức hao phí của dãy phép toán bằng cách định cận mức hao phí thực tế của từng phép toán riêng biệt, ta có thể dùng kỹ thuật phân tích khấu trừ để cung cấp một cận trên mức hao phí thực tế của nguyên cả dãy. Một lý do mà ý tưởng này có thể hiệu quả đó là trong một dãy phép toán, tất cả các phép toán riêng lẻ không thể chạy trong các cận thời gian ca xấu nhất đã biết của chúng. Tuy có vài phép toán tỏ ra tốn kém, song nhiều phép toán khác có thể rẻ. Tuy nhiên, phân tích khấu trừ không chỉ là một công cụ phân tích; nó còn là cách để nghĩ về thiết kế của các thuật toán, bởi thiết kế của một thuật toán và phân tích thời gian thực hiện của nó thường đan quyện mật thiết với nhau. Chương 18 sẽ giới thiệu ba cách tương đương để thực hiện một phân tích khấu trừ của một thuật toán.

16 Lập trình động

Giống như phương pháp chia để trị, lập trình động giải quyết các bài toán bằng cách tổ hợp các nghiệm của các bài toán con. ("Lập trình" trong ngữ cảnh này có nghĩa là một phương pháp lập bảng, chứ không phải viết mã máy tính.) Như đã thấy trong Chương 1, các thuật toán chia để trị phân hoạch bài toán thành các bài toán con độc lập, giải quyết các bài toán con một cách đệ quy, rồi tổ hợp các nghiệm của chúng để giải quyết bài toán ban đầu. Ngược lại, lập trình động có thể áp dụng khi các bài toán con không độc lập, nghĩa là, khi các bài toán con chia sẻ các bài toán "cháu" [subsubproblems]. Trong ngữ cảnh này, một thuật toán chia để trị thực hiện công việc nhiều hơn mức cần thiết, liên tục giải quyết các bài toán cháu chung. Một thuật toán lập trình động giải quyết mọi bài toán cháu nhất loạt rồi lưu đáp án vào một bảng, nhờ đó tránh được việc tính toán lại đáp án mọi lần gặp bài toán cháu.

Lập trình động thường được áp dụng cho ***các bài toán tối ưu hóa.*** Trong các bài toán như vậy thường có thể có nhiều giải pháp khả dĩ. Mỗi giải pháp có một giá trị, và ta muốn tìm một giải pháp có giá trị tối ưu (cực tiểu hoặc cực đại). Ta gọi kiểu giải pháp như vậy là *một* giải pháp tối ưu cho bài toán, trái với *tối* ưu nghiệm, bởi có thể có vài nghiệm đạt được giá trị tối ưu .

Sự phát triển của một thuật toán lập trình động có thể được tách nhỏ thành một dãy bốn bước.

1. Định rõ đặc điểm cấu trúc của một giải pháp tối ưu.
2. Định nghĩa theo đệ quy giá trị của một giải pháp tối ưu.
3. Tính toán giá trị của một giải pháp tối ưu theo kiểu dưới-lên.
4. Kiến tạo một giải pháp tối ưu từ thông tin đã tính toán.

Các bước 1-3 hình thành cơ sở của một giải pháp lập trình động cho một bài toán. Bước 4 có thể bỏ qua nếu chỉ cần giá trị của một giải pháp tối ưu. Khi phải thực hiện bước 4, đôi lúc ta duy trì thông tin bổ sung trong khi tính toán trong bước 3 để tạo thuận lợi cho việc kiến tạo một giải pháp tối ưu.

Các đoạn dưới đây dùng phương pháp lập trình động để giải quyết vài bài toán tối ưu hóa. Đoạn 16.1 yêu cầu cách nhân một chuỗi xích ma trận để có thể thực hiện tổng phép nhân vô hướng ít nhất. Căn cứ vào ví dụ này về lập trình động, Đoạn 16.2 mô tả hai đặc tính chính mà một bài toán phải có để lập trình động trở thành một kỹ thuật giải pháp có thể đứng vững. Sau đó, đoạn 16.3 nêu cách tìm dãy con chung dài nhất của hai dãy. Cuối cùng, Đoạn 16.4 sử dụng lập trình động để tìm một phép tam giác phân tối ưu của một đa giác lồi, một bài toán mà đnág ngạc nhiên làm sao lại tương tự như phép nhân xích ma trận.

16.1 Phép nhân xích ma trận

Ví dụ đầu tiên của chúng ta về lập trình động là một thuật toán giải quyết bài toán của phép nhân xích ma trận. Cho một dãy (xích) $<A_1, A_2, \ldots, A_n>$ ma trận sẽ được nhân, và ta muốn tính tích

$$A_1A_2 \ldots A_n . \qquad (16.1)$$

Có thể đánh giá biểu thức (16.1) bằng thuật toán chuẩn để nhân các cặp ma trận dưới dạng một chương trình con một khi ta đã ngoặc đơn nó để giải quyết tất cả mọi trường hợp tối nghĩa trong cách nhân các ma trận với nhau. Một tích của các ma trận ***được ngoặc đơn đầy đủ*** nếu nó là một ma trận đơn hoặc tích của hai tích ma trận được ngoặc đơn đầy đủ, được bao bởi các dấu ngoặc đơn. Phép nhân ma trận mang tính kết hợp, và do đó tất cả các phép ngoặc đơn đều cho ra cùng tích. Ví dụ, nếu xích của các ma trận là $<A_1, A_2, A_3, A_4>$, tích $A_1A_2A_3A_4$ có thể được ngoặc đơn đầy đủ theo năm cách riêng biệt:

$$(A_1A_2(A_3A_4))) ,$$
$$(A_1((A_2A_3)A_4)) ,$$
$$((A_1A_2)(A_3A_4)) ,$$
$$((A_1(A_2A_3))A_4) ,$$
$$(((A_1 A_2)A_3)A_4) .$$

Cách mà ta ngoặc đơn một chuỗi các ma trận có thể tác động đáng kể đến mức hao phí của việc đánh giá tích. Trước tiên, ta xem xét mức hao phí của việc nhân hai ma trận. Mã giả dưới đây cho ra thuật toán chuẩn. Các thuộc tính *rows* và *columns* là các số lượng hàng và cột trong một ma trận.

```
MATRIX-MULTIPLY(A, B)
1 nếu columns[A] ≠ rows[B]
2     then error "incompatible dimensions"
3     else for i ← 1 to rows[A]
4              do for j ← 1 to columns[B]
5                     do C[i, j] ← 0
6                        for k ← 1 to columns[A]
7                            do C[i, j] ← C[i, j] + A[i, k] - B[k, j]
8          return C
```

Ta chỉ có thể nhân hai ma trận A và B nếu số lượng cột của A bằng số lượng hàng của B. Nếu A là một ma trận $p \times q$ và B là một ma trận $q \times r$, ma trận kết quả C là một ma trận $p \times r$. Thời gian để tính toán C bị chi phối bởi số lượng các phép nhân vô hướng trong dòng 7, là pqr. Dưới đây, ta sẽ diễn tả thời gian thực hiện theo dạng số lượng các phép nhân vô hướng.

Để minh họa các mức hao phí khác nhau mà các phép ngoặc đơn khác nhau của một tích ma trận phải gánh, ta xét bài toán của một xích $<A_1, A_2, A_3>$ gồm ba ma trận. Giả sử các chiều của các ma trận là 10×100, 100×5, và 5×50, theo thứ tự nêu trên. Nếu nhân theo phép ngoặc đơn $((A_1A_2)A_3)$, ta thực hiện $10 \cdot 100 \cdot 5 = 5000$ phép nhân vô hướng để tính toán 10×5 tích ma trận $A_1 A_2$, cộng với một $10 \cdot 5 \cdot 50 = 2500$ phép nhân vô hướng khác để nhân ma trận này với A_3, cho một tổng 7500 phép nhân vô hướng. Thay vì thế, nếu nhân theo phép ngoặc đơn $(A_1 (A_2A_3))$, ta thực hiện $100 \cdot 5 \cdot 50 = 25{,}000$ phép nhân vô hướng để tính toán 100×50 tích ma trận A_2A_3, cộng với một $10 \cdot 100 \cdot 50 = 50{,}000$ phép nhân vô hướng khác để nhân A_1 với ma trận này, cho một tổng 75,000 phép nhân vô hướng. Như vậy, tính toán tích theo phép ngoặc đơn đầu tiên sẽ nhanh hơn gấp 10 lần.

Có thể phát biểu ***bài toán nhân xích ma trận*** như sau: cho một xích $(A_1, A_2, \ldots, A_n)$ n ma trận, ở đó với $i = 1, 2, \ldots, n$, ma trận A_i has chiều $p_{i-1} \times p_i$ ngoặc đơn đầy đủ tích $A_1A_2 \ldots A_n$ sao cho có thể giảm thiểu số lượng phép nhân vô hướng.

Đếm số lượng các phép ngoặc đơn

Trước khi giải quyết bài toán nhân xích ma trận bằng lập trình động, ta nên tự hiểu rằng việc kiểm tra thấu đáo tất cả các phép ngoặc đơn khả dĩ không mang lại một thuật toán hiệu quả. Hãy nêu rõ số lượng các phép ngoặc đơn thay thế của một dãy n ma trận theo $P(n)$. Bởi ta có

thể tích một dãy n ma trận giữa các ma trận thứ k và thứ $(k + 1)$ với bất kỳ $k = 1, 2, \ldots, n - 1$ rồi ngoặc đơn hai dãy con kết quả một cách độc lập, ta được phép truy toán

$$P(n) = \begin{cases} 1 & \text{nếu } n = 1 \,, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{nếu } n \geq 2 \,. \end{cases}$$

Bài toán 13-4 yêu cầu bạn chứng tỏ giải pháp của phép truy toán này là dãy ***các số Catalan***:

$$P(n) = C(n-1) \,,$$

ở đó

$$C(n) = \frac{1}{n+1}\binom{2n}{n} = \Omega(4^n/n^{3/2}) \,.$$

Như vậy, số lượng các giải pháp là theo số mũ trong n, và do đó phương pháp cưỡng bức của đợt tìm kiếm thấu đáo là một chiến lược tồi để xác định phép ngoặc đơn tối ưu của một ma trận xích.

Cấu trúc của một phép ngoặc đơn tối ưu

Bước đầu tiên của kiểu mẫu lập trình động đó là định rõ đặc điểm cấu trúc của một giải pháp tối ưu. Với bài toán phép nhân xích ma trận, ta có thể thực hiện bước này như sau. Để tiện dụng, ta chấp nhận hệ ký hiệu $A_{i..j}$ với ma trận là kết quả của việc đánh giá tích $A_iA_{i+1} \ldots A_j$. Một phép ngoặc đơn tối ưu của tích $A_1 A_2 \ldots A_n$ sẽ tách tích giữa A_k và A_{k+1} với một số nguyên k trong miền giá trị $1 \leq k < n$. Nghĩa là, với một giá trị nào đó của k, trước tiên ta tính toán các ma trận $A_{1..k}$ và $A_{k+1..n}$ rồi nhân chúng với nhau để tạo ra tích chung cuộc $A_{1..n}$. Như vậy, mức hao phí của phép ngoặc đơn tối ưu này sẽ là mức hao phí để tính toán ma trận $A_{1..k}$, cộng với mức hao phí để tính toán $A_{k+1..n}$, cộng với mức hao phí để nhân chúng với.

Nhận xét chính đó là phép ngoặc đơn của xích con "tiền tố" $A_1 A_2 \ldots A_k$ trong phép ngoặc đơn tối ưu này của $A_1 A_2 \ldots A_n$ phải là một phép ngoặc đơn *tối ưu* của $A_1A_2 \ldots A_k$. Tại sao? Nếu có một cách ít hao phí hơn để ngoặc đơn $A_1 A_2 \ldots A_k$, thì việc thay phép ngoặc đơn đó trong phép ngoặc đơn tối ưu của $A_1 A_2 \ldots A_k$ sẽ tạo ra một phép ngoặc đơn khác của $A_1 A_2 \ldots A_n$ mà mức hao phí của nó thấp hơn mức tối ưu: một sự mâu thuẫn. Nhận xét tương tự cũng áp dụng cho phép ngoặc đơn của xích con $A_{k+1}A_{k+2} \ldots A_n$ trong phép ngoặc đơn tối ưu của $A_1A_2 \ldots A_n$ nó phải là một phép ngoặc đơn tối ưu của $A_{k+1} A_{k+2} \ldots A_n$.

Như vậy, một giải pháp tối ưu cho trường hợp bài toán phép nhân xích ma trận sẽ chứa trong nó các giải pháp tối ưu đối với các trường hợp bài toán con. Cấu trúc con tối ưu trong một giải pháp tối ưu là một trong các dấu xác nhận chất lượng về khả năng áp dụng của phương pháp lập trình động, như sẽ thấy trong Đoạn 16.2.

Một giải pháp đệ quy

Bước thứ hai của kiểu mẫu lập trình động đó là định nghĩa giá trị của một giải pháp tối ưu theo đệ quy dưới dạng các nghiệm tối ưu cho các bài toán con. Với bài toán nhân xích ma trận, ta chọn cho các bài toán con của chúng ta các bài toán xác định mức hao phí cực tiểu của một phép ngoặc đơn của $A_iA_{i+1}...A_j$ với $1 \le i \le j \le n$. Cho $m[i, j]$ là số lượng cực tiểu các phép nhân vô hướng cần có để tính toán ma trận $A_{i..j}$; thì mức hao phí của cách rẻ nhất để tính toán $A_{1..n}$ sẽ là $m[1, n]$.

Ta có thể định nghĩa $m[i, j]$ một cách đệ quy như sau. Nếu $i = j$, xích chỉ bao gồm một ma trận $A_{i..i} = A_i$, do đó không cần phép nhân vô hướng nào để tính toán tích. Như vậy, $m[i, i] = 0$ với $i = 1, 2,..., n$. Để tính $m[i, j]$ khi $i < j$, ta vận dụng cấu trúc của một giải pháp tối ưu trong bước 1. Hãy mặc nhận phép ngoặc đơn tối ưu tách tích $A_iA_{i+1} ... A_j$ giữa A_k và A_{k+1} ở đó $i \le k < j$. Thì, $m[i, j]$ bằng mức hao phí cực tiểu để tính toán các tích con $A_{i..k}$ và $A_{k+1..j}$, cộng mức hao phí của việc nhân hai ma trận này với nhau. Bởi tiến trình tính toán tích ma trận $A_{i..k}A_{k+i..j}$ thực hiện $P_{i-1}p_kp_j$ phép nhân vô hướng, nên ta có

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j \,.$$

Phương trình đệ quy này mặc nhận ta biết giá trị của k, mà ta không biết. Tuy nhiên, chỉ có $j - i$ giá trị khả dĩ cho k, tức $k = i, i + 1,..., j - 1$. Bởi phép ngoặc đơn tối ưu phải dùng một trong các giá trị này cho k, nên ta chỉ cần kiểm tra tất cả để tìm ra cái tốt nhất. Như vậy, phần định nghĩa đệ quy của chúng ta đối với mức hao phí cực tiểu của việc ngoặc đơn tích $A_iA_{i+1} ... A_j$ trở thành

$$P(n) = \begin{cases} 0 & \text{nếu } i = j\,, \\ \min\limits_{i \le k < j} \{m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j\} & \text{nếu } i < j\,. \end{cases} \quad (16.2)$$

Các giá trị $m[i, j]$ cho các mức hao phí của các giải pháp tối ưu đối với các bài toán con. Để giúp theo dõi cách kiến tạo một giải pháp tối ưu, ta hãy định nghĩa $s[i, j]$ là một giá trị của k tại đó ta có thể tách tích $A_iA_{i+1} ... A_j$ để được một phép ngoặc đơn tối ưu. Nghĩa là, $s[i, j]$ bằng một giá trị k sao cho $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$.

Tính toán các mức hao phí tối ưu

Vào lúc này, ta có thể đơn giản viết một thuật toán đệ quy dựa trên phép truy toán (16.2) để tính toán mức hao phí cực tiểu $m[1, n]$ để nhân $A_1 A_2 \ldots A_n$. Tuy nhiên, như sẽ thấy trong Đoạn 16.2, thuật toán này mất thời gian mũ—không tốt hơn phương pháp cưỡng bức để kiểm tra từng cách ngoặc đơn tích.

Nhận xét quan trọng có thể rút ra vào lúc này đó là ta có tương đối ít bài toán con: một bài toán cho mỗi chọn lựa của i và j thỏa $1 \le i \le j \le n$, hoặc $\binom{n}{2} + n = \Theta(n^2)$ tính tổng cộng. Một thuật toán đệ quy có thể gặp từng bài toán con nhiều lần trong các nhánh khác nhau của cây đệ quy. Tính chất phủ chồng các bài toán con này là dấu chất lượng thứ hai về khả năng áp dụng lập trình động.

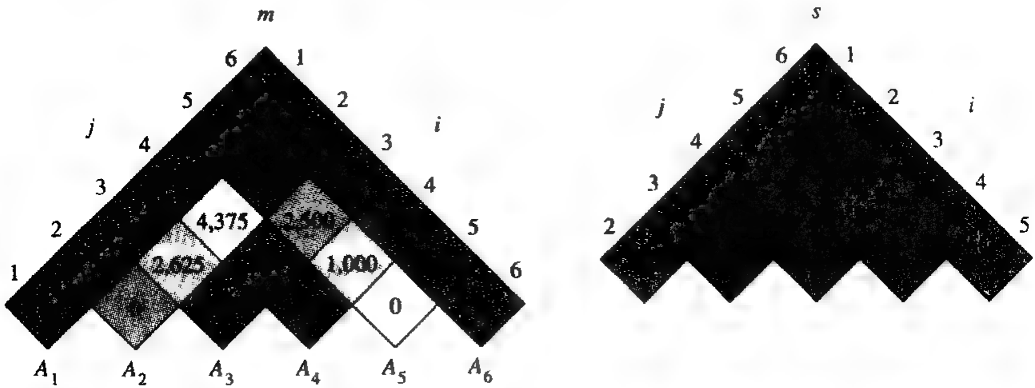
Thay vì tính toán giải pháp cho phép truy toán (16.2) theo đệ quy, ta thực hiện bước thứ ba của kiểu mẫu lập trình động và tính toán mức hao phí tối ưu bằng cách dùng cách tiếp cận dưới-lên. Mã giả dưới đây mặc nhận ma trận A_i có các chiều $p_{i-1} \times p_i$ với $i = 1, 2, \ldots, n$. Nhập liệu là một dãy $\langle p_0, p_1, \ldots, p_n \rangle$, ở đó *length*$[p] = n + 1$. Thủ tục sử dụng một bảng phụ $m[1..n, 1..n]$ để lưu trữ $m[i, j]$ mức hao phí và một bảng phụ $s[1..n, 1..n]$ ghi nhận chỉ số nào của k đã đạt được mức hao phí tối ưu trong tính toán $m[i, j]$.

```
MATRIX-CHAIN-ORDER(p)
n ← length[p] - 1
for i ← 1 to n
      do m[i, i] ← 0
for l ← 2 to n
      do for i ← 1 to n - l + 1
               do j ← i + l - 1
                    m[i, j] ← ∞
                    for k ← i to j - 1
                         do q ← m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j
                              if q < m[i, j]
                                 then m[i, j] ← q
                                      s[i, j] ← k
return m and s
```

Thuật toán điền vào bảng m theo cách tương ứng với việc giải quyết bài toán phép ngoặc đơn trên các xích ma trận có chiều dài gia tăng.

Phương trình (16.2) cho thấy mức hao phí $m[i, j]$ của việc tính toán một tích xích ma trận của $j - i + 1$ ma trận chỉ tùy thuộc vào các mức hao phí khi tính toán các tích xích ma trận của ít hơn $j - i + 1$ ma trận. Nghĩa là, với $k = i, i + 1,..., j - 1$, ma trận $A_{i..k}$ là một tích của $k - i + 1 < j - i + 1$ ma trận và ma trận $A_{k+1..j}$ là một tích của $j - k < j - i + 1$ ma trận.

Trước tiên, thuật toán sẽ tính $m[i, i] \leftarrow 0$ với $i = 1, 2,..., n$ (mức hao phí cực tiểu cho các xích có chiều dài 1) trong các dòng 2-3. Sau đó, nó sử dụng phép truy toán (16.2) để tính toán $m[i, i + 1]$ với $i = 1, 2,..., n - 1$ (mức hao phí cực tiểu cho các xích có chiều dài $l = 2$) trong lần thi hành đầu tiên của vòng lặp trong các dòng 4-12. Lần thứ hai qua vòng lặp, nó tính toán $m[i, i + 2]$ với $i = 1, 2,..., n - 2$ (mức hao phí cực tiểu cho các xích có chiều dài $l = 3$), vân vân. Tại mỗi bước, $m[i, j]$ mức hao phí đã tính toán trong các dòng 9-12 chỉ tùy thuộc vào các khoản nhập bảng $m[i, k]$ và $m[k + 1, j]$ đã tính toán sẵn.



Hình 16.1 Các bảng *m* và *s* được tính toán bởi MATRIX-CHAIN-ORDER với $n = 6$ và các chiều ma trận dưới đây:

ma trận	chiều
A_1	30 x 35
A_2	35 x 15
A_3	15 x 5
A_4	5 x 10
A_5	10 x 20
A_6	20 x 25

Các bảng được quay để đường chéo chính chạy ngang. Chỉ đường chép chính và tam giác phía trên được dùng trong bảng *m*, và chỉ tam giác phía trên được dùng trong bảng *s*. Số lượng cực tiểu các phép nhân vô hướng để nhân 6 ma trận là $m[1,6] = 15,125$. Trong số các khoản

nhập tô bóng sáng, các cặp có cùng kiểu tô bóng được gom với nhau trong dòng 9 khi tính toán

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375 \end{cases}$$

$$= 7125 .$$

Hình 16.1 minh họa thủ tục này trên một xích gồm $n = 6$ ma trận. Do ta đã định nghĩa $m[i, j]$ với chỉ $i \le j$, nên thủ tục chỉ dùng phần bảng m nằm hẳn phía trên đường chéo chính. Hình nêu bảng đã quay để tạo đường chéo chính chạy ngang. Xích ma trận được liệt kê dọc theo đáy. Dùng bố cục này, mức hao phí cực tiểu $m[i, j]$ để nhân một chuỗi con $A_i A_{i+1} \ldots A_j$ gồm các ma trận có thể tìm thấy tại giao điểm của các dòng chạy hướng đông bắc từ A_i và hướng tây bắc từ A_j. Mỗi hàng ngang trong bảng chứa các khoản nhập cho các xích ma trận có cùng chiều dài. MATRIX-CHAIN-ORDER tính toán các hàng từ dưới lên và từ trái sang phải trong mỗi hàng. Một khoản nhập $m[i, j]$ được tính toán bằng các tích $p_{i-1} p_k p_j$ với $k = i, i + 1, \ldots, j - 1$ và tất cả các khoản nhập hướng tây nam và đông nam từ $m[i, j]$.

Một đợt thẩm tra đơn giản về cấu trúc vòng lặp lồng ghép của MATRIX-CHAIN-ORDER cho ra một thời gian thực hiện $O(n^3)$ cho thuật toán. Các vòng lặp được lồng sâu ba cấp, và mỗi chỉ số vòng lặp (l, i, và k) đảm nhiệm tối đa n giá trị. Bài tập 16.1-3 yêu cầu bạn chứng tỏ thời gian thực hiện của thuật toán này thực tế cũng là $\Omega(n^3)$. Thuật toán yêu cầu $\Theta(n^2)$ không gian để lưu trữ các bảng m và s. Như vậy, MATRIX-CHAIN-ORDER hiệu quả hơn nhiều so với phương pháp thời gian mũ để điểm danh tất cả các phép ngoặc đơn khả dĩ và kiểm tra từng phép một.

Kiến tạo một giải pháp tối ưu

Tuy MATRIX-CHAIN-ORDER xác định số lượng tối ưu các phép nhân vô hướng cần có để tính toán một tích xích ma trận, song nó không trực tiếp nêu cách nhân các ma trận. Bước 4 của kiểu mẫu lập trình động đó là kiến tạo một giải pháp tối ưu từ thông tin đã tính toán.

Trong trường hợp này, ta dùng bảng $s[1..n, 1..n]$ để xác định cách tốt nhất để nhân các ma trận. Mỗi khoản nhập $s[i, j]$ ghi nhận giá trị của k để phép ngoặc đơn tối ưu $A_i A_{i+1} \ldots A_j$ tách tích giữa A_k và A_{k+1}. Như vậy, ta biết rằng phép nhân ma trận chung cuộc trong khi tính toán $A_{1..n}$ tối ưu là $A_{1..s[1,n]} A_{s[1,n]+1..n}$. Các phép nhân ma trận trước đó có thể được tính toán theo đệ quy, bởi $s[1, s[1,n]]$ xác định phép nhân ma trận cuối trong khi

tính toán $A_{1..s[1,n]}$, và $s[s[1, n] + 1, n]$ xác định phép nhân ma trận cuối trong khi tính toán $A_{s[1,n]+1..n}$. Thủ tục đệ quy dưới đây tính toán tích của xích ma trận $A_{i..j}$ căn cứ vào các ma trận $A = <A_1, A_2,..., A_n>$, bảng s mà MATRIX-CHAIN-ORDER đã tính, và các chỉ số i và j. Lệnh gọi ban dầu là MATRIX-CHAIN-MULTIPLY(A, s, 1, n).

MATRIX-CHAIN-MULTIPLY (A, s, i, j)

1 **if** $j > i$

2 **then** $X \leftarrow$ MATRIX-CHAIN-MULTIPLY($A, s, i, s[i, j]$)

3 $Y \leftarrow$ MATRIX-CHAIN-MULTIPLY($A, s, s[i, j] + 1, j$)

4 **trả về** MATRIX-MULTIPLY(X, Y)

5 **else return** A_i

Trong ví dụ của Hình 16.1, lệnh gọi MATRIX-CHAIN-MULTIPLY(A, s, 1, 6) tính toán tích xích ma trận theo phép ngoặc đơn

$$((A_1(A_2A_3))((A_4A_5)A_6)) \qquad (16.3)$$

Bài tập

16.1-1

Tìm một phép ngoặc đơn tối ưu của một tích xích ma trận có dãy các chiều là <5,10, 3, 12, 5, 50, 6>.

16.1-2

Nêu một thuật toán hiệu quả PRINT-OPTIMAL-PARENS để in phép ngoặc đơn tối ưu của một xích ma trận căn cứ vào bảng s mà MATRIX-CHAIN-ORDER đã tính. Phân tích thuật toán.

16.1-3

Cho $R(i, j)$ là số lần mà MATRIX-CHAIN-ORDER tham chiếu khoản nhập bảng $m[i, j]$ trong khi tính toán các khoản nhập bảng khác. Chứng tỏ tổng số các tham chiếu của nguyên cả bảng là

$$\sum_{i=1}^{n}\sum_{j=i}^{n} R(i, j) = \frac{n^3 - n}{3}.$$

(*Mách nước*: Bạn có thể thấy đồng nhất thức $\sum_{i=1}^{n} i^2 = n(n + 1)(2n + 1)/6$ là hữu ích.)

16.1-4

Chứng tỏ một phép ngoặc đơn đầy đủ của một biểu thức n-thành phần có chính xác n - 1 cặp các dấu ngoặc đơn.

16.2 Các thành phần của lập trình động

Tuy chỉ mới làm việc qua một ví dụ về phương pháp lập trình động, song có thể bạn vẫn tự hỏi khi nào ta áp dụng phương pháp này. Trên quan điểm thiết kế kỹ thuật, khi nào ta phải tìm một giải pháp lập trình động cho một bài toán? Trong đoạn này, ta xem xét hai thành tố chính mà bài toán tối ưu hóa phải có để có thể áp dụng phương pháp lập trình động: cấu trúc con tối ưu và các bài toán con phủ chồng. Ta cũng xem xét một phương pháp biến thể, có tên "phép memo" [memoization], để vận dụng tính chất của các bài toán con phủ chồng.

Cấu trúc con tối ưu

Bước đầu tiên trong tiến trình giải quyết một bài toán tối ưu hóa bằng lập trình động đó là định rõ đặc điểm cấu trúc của một giải pháp tối ưu. Ta nói rằng một bài toán biểu lộ ***cấu trúc con tối ưu*** [optimal substructure] nếu một giải pháp tối ưu cho bài toán chứa trong nó các giải pháp tối ưu cho các bài toán con. Mỗi khi một bài toán biểu lộ cấu trúc con tối ưu, nó là manh mối tốt cho biết có thể áp dụng phương pháp lập trình động. (Tuy nhiên, như vậy cũng có nghĩa là có thể áp dụng một chiến lược tham [greedy]. Xem Chương 17.)

Đoạn 16.1 đã cho biết bài toán của phép nhân xích ma trận biểu lộ cấu trúc con tối ưu. Ta đã nhận xét rằng một phép ngoặc đơn tối ưu của $A_1 A_2 \ldots A_n$ tách tích giữa A_k và A_{k+1} chứa trong nó các giải pháp tối ưu cho các bài toán ngoặc đơn $A_1A_2 \ldots A_k$ và $A_{k+1}A_{k+2} \ldots A_n$. Kỹ thuật mà ta đã dùng để chứng tỏ các bài toán con có các giải pháp tối ưu là điển hình. Ta mặc nhận rằng có một giải pháp tốt hơn cho bài toán con và nêu giả thiết này mâu thuẫn ra sao đối với khả năng tối ưu của giải pháp cho bài toán ban đầu.

Cấu trúc con tối ưu của một bài toán thường gợi ý một không gian thích hợp của các bài toán con ở đó có thể áp dụng phương pháp lập trình động. Thông thường, có vài lớp bài toán con mà ta có thể xem là "tự nhiên" cho một bài toán. Ví dụ, không gian của các bài toán con mà ta đã xét cho phép nhân xích ma trận có chứa tất cả các xích con của xích nhập liệu. Ngoài ra, ta cũng có thể chọn các dãy ma trận tùy ý từ xích nhập liệu làm không gian các bài toán con, song không gian các bài toán con này không nhất thiết phải lớn. Một thuật toán lập trình động dựa trên không gian các bài toán con này sẽ giải quyết thêm nhiều bài toán hơn so với mức bắt buộc.

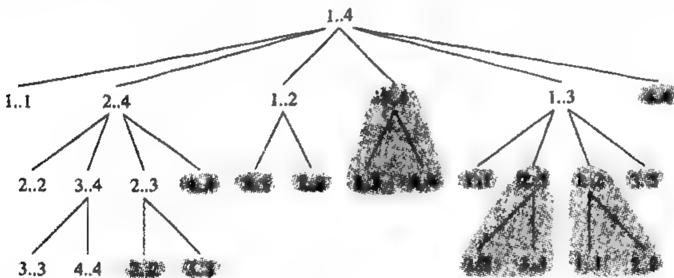
Tiến trình thẩm tra cấu trúc con tối ưu của một bài toán lặp lại trên các trường hợp bài toán con là một cách tốt để suy ra một không gian các bài toán con thích hợp cho phương pháp lập trình động. Ví dụ, sau

khi xem xét cấu trúc của một giải pháp tối ưu cho một bài toán xích ma trận, ta có thể lặp lại và xem xét cấu trúc của các giải pháp tối ưu cho các bài toán con, các bài toán cháu, vân vân. Ta thấy rằng tất cả các bài toán con bao gồm các xích con của $<A_1, A_2,..., A_n>$. Như vậy, tập hợp các xích theo dạng $(A_i, A_{i+1},..., A_j)$ với $1 \leq i \leq j \leq n$ sẽ tạo một không gian hợp lý và tự nhiên của các bài toán con để sử dụng.

Các bài toán con phủ chồng

Thành tố thứ hai mà một bài toán tối ưu hóa phải có để có thể áp dụng kỹ thuật lập trình động đó là không gian các bài toán con phải "nhỏ" theo nghĩa là một thuật toán đệ quy cho bài toán sẽ giải quyết lặp lại các bài toán con tương tự, thay vì luôn phát sinh các bài toán con mới. Thông thường, tổng số các bài toán con riêng biệt là một đa thức có kích cỡ nhập liệu. Khi một thuật toán đệ quy ghé thăm hoài cùng một bài toán, ta nói rằng bài toán tối ưu hóa có ***các bài toán con phủ chồng*** [overlapping subproblems]. Ngược lại, một bài toán thích hợp với cách tiếp cận chia để trị thường phát sinh các bài toán hoàn toàn mới tại mỗi bước của đệ quy. Các thuật toán lập trình động thường vận dụng các bài toán con phủ chồng bằng cách giải quyết từng bài toán con một rồi lưu trữ giải pháp trong một bảng ở đó nó có thể được tra cứu khi cần, dùng thời gian bất biến cho mỗi lần tra cứu.

Để minh họa tính chất các bài toán con phủ chồng, ta hãy xem xét lại bài toán nhân xích ma trận. Xem lại Hình 16.1, ta nhận thấy MATRIX-CHAIN-ORDER tra cứu lặp lại giải pháp cho các bài toán con trong các hàng dưới khi giải quyết các bài toán con trong các hàng trên. Ví dụ, khoản nhập $m[3, 4]$ được tham chiếu 4 lần: trong các lần tính toán $m[2, 4]$, $m[1, 4]$, $m[3, 5]$, và $m[3, 6]$. Nếu $m[3, 4]$ được tính lại mỗi lần, thay vì được tra cứu, thời gian thực hiện sẽ tăng lên đáng kể. Để xem điều này, ta xét thủ tục đệ quy (không hiệu quả) dưới đây xác định $m[i, j]$, số lượng cực tiểu các phép nhân vô hướng cần có để tính toán tích xích ma trận $A_{i..j} = A_iA_{i+1} \dots A_j$. Thủ tục trực tiếp dựa trên phép truy toán (16.2).



Hình 16.2 Cây đệ quy để tính toán RECURSIVE-MATRIX-CHAIN(p, 1, 4). Mỗi nút chứa các tham số i và j. Các lần tính toán được thực hiện trong một cây con có tô bóng được thay bởi một đợt tra cứu bảng đơn lẻ trong MEMOIZED-MATRIX-CHAIN(p, 1, 4).

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
1 if i = j
2     then return 0
3 m[i, j] ← ∞
4 for k ← i to j – 1
5         do q ← RECURSIVE-MATRIX-CHAIN(p, d, k)
                 + RECURSIVE-MATRIX-CHAIN(p, k + 1, j) + p_{i-1} p_k p_j
6             if q < m[i, j]
7                 then m[i, j] ← q
8 return m[i, j]
```

Hình 16.2 nêu cây đệ quy được tạo bởi lệnh gọi RECURSIVE-MATRIX-CHAIN(p, 1, 4). Mỗi nút được gán nhãn bằng các giá trị của các tham số i và j. Nhận thấy có vài cặp giá trị xảy ra nhiều lần.

Thực vậy, ta có thể chứng tỏ thời gian thực hiện $T(n)$ để tính toán $m[1, n]$ bằng thủ tục đệ quy này ít nhất là hàm số mũ trong n. Ta hãy mặc nhận việc thi hành các dòng 1-2 và các dòng 6-7, mỗi lần mất ít nhất thời gian đơn vị. Việc thẩm tra thủ tục cho ra phép truy toán

$$T(1) \geq 1 ,$$

$$T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \qquad \text{với } n > 1 .$$

Lưu ý, với $i = 1, 2, \ldots, n$-1, mỗi số hạng $T(i)$ xuất hiện một lần dưới dạng $T(k)$ và một lần dưới dạng $T(n - k)$, và tập hợp n - 1 các 1 trong phép lấy tổng chung với 1 ra phía trước, ta có thể viết lại phép truy toán như sau

$$T(n) \geq 2 \sum_{i=1}^{n-1} T(i) + n . \qquad (16\text{-}4)$$

Ta sẽ chứng minh $T(n) = \Omega(2^n)$ bằng phương pháp thay thế. Cụ thể, ta sẽ chứng tỏ $T(n) \geq 2^{n-1}$ với tất cả $n \geq 1$. Cơ bản là dễ, bởi $T(1) \geq 1 = 2^0$. Theo quy nạp, với $n \geq 2$ ta có

$$\begin{aligned} T(n) &\geq 2 + \sum_{i=1}^{n-1} 2^{i-1} + n \\ &= 2 + \sum_{i=1}^{n-2} 2^i + n \\ &= 2\,(2^{n-1} - 1) + n \end{aligned}$$

$= (2^n - 2) + n$

$\geq 2^{n-1}$,

hoàn tất phần chứng minh. Như vậy, tổng số công việc được thực hiện bởi lệnh gọi RECURSIVE-MATRIX-CHAIN(p, 1, n) ít nhất là hàm số mũ trong n.

So sánh thuật toán đệ quy trên xuống này với thuật toán lập trình động dưới-lên. Thuật toán sau tỏ ra hiệu quả hơn bởi nó vận dụng tính chất của các bài toán con phủ chồng. Chỉ có $\Theta(n^2)$ bài toán con khác nhau, và thuật toán lập trình động giải quyết từng bài toán con này chính xác một lần. Trong khi đó, thuật toán đệ quy phải giải quyết lặp lại từng bài toán con mỗi lần nó tái xuất hiện trong cây đệ quy. Mỗi khi một cây đệ quy cho giải pháp đệ quy tự nhiên đối với một bài toán chứa cùng bài toán con lặp lại, và tổng số các bài toán con khác nhau không lớn, tốt nhất ta nên xem có thể vận dụng phương pháp lập trình động hay không.

Phép memo

Có một biến thể của kỹ thuật lập trình động thường cung cấp hiệu năng của cách tiếp cận lập trình động bình thường trong khi vẫn duy trì một chiến lược trên xuống. Ý tưởng đó là ***memo hóa*** [memoize] thuật toán đệ quy tự nhiên song không hiệu quả. Cũng như trong lập trình động bình thường, ta duy trì một bảng có các giải pháp bài toán con, nhưng cấu trúc điều khiển để điền bảng lại giống với thuật toán đệ quy hơn.

Một thuật toán đệ quy memo duy trì một khoản nhập trong một bảng cho giải pháp của mỗi bài toán con. Thoạt đầu, mỗi khoản nhập bảng chứa một giá trị đặc biệt để nêu rõ khoản nhập phải được điền. Khi gặp bài toán con lần đầu tiên trong khi thi hành thuật toán đệ quy, giải pháp của nó được tính toán rồi được lưu trữ trong bảng. Mỗi lần gặp bài toán con sau đó, giá trị đã lưu trữ trong bảng đơn giản được tra cứu và trả về[1].

Thủ tục dưới đây là một phiên bản memo của RECURSIVE-MATRIX-CHAIN.

MEMOIZED-MATRIX-CHAIN(p)

1 Cách tiếp cận này hàm ý đã biết tập hợp tất cả các tham số bài toán con khả dĩ và đã thiết lập quan hệ giữa các vị trí bảng và các bài toán con. Một cách tiếp cận khác đó là memo hóa bằng cách dùng kỹ thuật ánh số với các tham số bài toán con dưới dạng các khóa.

```
1  n ← length[p] - 1
2  for i ← 1 to n
3       do for j ← i to n
4            do m[i, j] ← ∞
5  return LOOKUP-CHAIN(p, 1, n)
LOOKUP-CHAIN(p, i, j)
1  if m[i, j] < ∞
2       then return m[i, j]
3  if i=j
4       then m[i, j] ← 0
5       else for k ← i to j - 1
6                do q ← LOOKUP-CHAIN(p, i, k)
                        + LOOKUP-CHAIN(p, k + 1, j) + p_{i-1} p_k p_j
7                   if q < m[i, j]
8                       then m[i, j] ← q
9  return m[i, j]
```

MEMOIZED-MATRIX-CHAIN, cũng như MATRIX-CHAIN-ORDER, duy trì một bảng $m[1..n, 1..n]$ gồm các giá trị đã tính toán của $m[i, j]$, số lượng cực tiểu các phép nhân vô hướng cần có để tính toán ma trận $A_{i..j}$. Thoạt đầu, mỗi khoản nhập bảng chứa giá trị ∞ để nêu rõ khoản nhập phải được điền. Khi thi hành lệnh gọi LOOKUP-CHAIN(p, i, j), nếu $m[i, j] < \infty$ trong dòng 1, thủ tục đơn giản trả về mức hao phí đã tính toán trước đó $m[i, j]$ (dòng 2). Bằng không, mức hao phí được tính toán như trong RECURSIVE-MATRIX-CHAIN, được lưu trữ trong $m[i, j]$, và được trả về. (Giá trị ∞ tỏ ra tiện dụng khi sử dụng cho một khoản nhập bảng chưa điền bởi nó là giá trị được dùng để khởi tạo $m[i, j]$ trong dòng 3 của RECURSIVE-MATRIX-CHAIN.) Như vậy, LOOKUP-CHAIN(p, i, j) luôn trả về giá trị của $m[i, j]$, nhưng chỉ tính toán nó nếu đây là lần đầu tiên LOOKUP-CHAIN được gọi với các tham số i và j.

Hình 16.2 minh họa cách thức mà MEMOIZED-MATRIX-CHAIN tiết kiệm thời gian trên RECURSIVE-MATRIX-CHAIN. Các cây con tô bóng biểu diễn các giá trị được tra cứu thay vì được tính toán.

Cũng như thuật toán lập trình động MATRIX-CHAIN-ORDER, thủ tục MEMOIZED-MATRIX-CHAIN chạy trong $O(n^3)$ thời gian. Mỗi trong số $\Theta(n^2)$ khoản nhập bảng được khởi tạo một lần trong dòng 4 của MEMOIZED-MATRIX-CHAIN và được điền vĩnh viễn bằng chỉ một

lệnh gọi LOOKUP-CHAIN. Mỗi trong số $\Theta(n^2)$ lệnh gọi này đến LOOKUP-CHAIN sẽ mất $O(n)$ thời gian, loại trừ thời gian bỏ ra để tính toán các khoản nhập bảng khác, do đó mất chung một tổng là $O(n^3)$. Như vậy, phép memo chuyển một thuật toán $\Omega(2^n)$ thành một thuật toán $O(n^3)$.

Tóm lại, bài toán nhân xích ma trận có thể được giải quyết trong $O(n^3)$ thời gian bằng một thuật toán memo trên xuống, hoặc một thuật toán lập trình động dưới lên. Cả hai phương pháp đều vận dụng tính chất các bài toán con phủ chồng. Tổng cộng chỉ có $\Theta(n^2)$ bài toán con khác nhau, và một trong hai phương pháp này sẽ tính toán giải pháp cho mỗi bài toán con một lần. Nếu không có phép memo, thuật toán đệ quy tự nhiên chạy trong thời gian mũ, bởi các bài toán con đã giải quyết lại được giải quyết lặp.

Theo cách làm chung, nếu tất cả bài toán con phải được giải quyết ít nhất một lần, một thuật toán lập trình động dưới lên thường làm tốt hơn thuật toán memo trên xuống theo một thừa số bất biến, bởi không có phần việc chung cho đệ quy và ít phần việc chung hơn để duy trì bảng. Ngoài ra, có một số bài toán mà ta có thể khai thác khuôn mẫu bình thường của các đợt truy cập bảng trong thuật toán lập trình động để rút gọn hơn nữa các yêu cầu về thời gian hoặc không gian. Mặt khác, nếu có vài bài toán con trong không gian bài toán con không cần giải quyết gì cả, giải pháp memo có ưu thế đó là chỉ giải quyết những bài toán con nào được yêu cầu cụ thể.

Bài tập

16.2-1

So sánh phép truy toán (16.4) với phép truy toán (8.4) nảy sinh trong khi phân tích thời gian thực hiện ca trung bình của kỹ thuật sắp xếp nhanh. Giải thích theo trực giác tại sao các giải pháp cho hai phép truy toán lại khác nhau đến thế.

16.2-2

Nêu một cách hiệu quả hơn để xác định số lượng tối ưu các phép nhân trong một bài toán phép nhân xích ma trận: liệt kê tất cả các cách ngoặc đơn tích và tính toán số lượng phép nhân cho từng cách, hoặc chạy RECURSIVE-MATRIX-CHAIN. Xác minh đáp án.

16.2-3

Vẽ cây đệ quy cho thủ tục MERGE-SORT từ Đoạn 1.3.1 trên một mảng gồm 16 thành phần. Giải thích tại sao phép memo lại không hiệu

quả trong việc tăng tốc một thuật toán chia để trị tốt như MERGE-SORT.

16.3 Dãy con chung dài nhất

Bài toán kế tiếp mà ta sẽ xét đó là bài toán dãy con chung dài nhất [longest common subsequence]. Một dãy con của một dãy đã cho chẳng qua là dãy đã cho với vài thành phần (có thể không có thành phần nào) được xóa. Chính thức, cho một dãy $X = <x_1, x_2,..., x_m>$, một dãy khác $Z = <z_1, z_2,..., z_k>$ là một ***dãy con*** của X nếu ở đó tồn tại một dãy tăng ngặt $<i_1 , i_2, ... , i_k >$ gồm các chỉ số của X sao cho với tất cả $j = 1, 2,..., k$, ta có $x_{i_j} = z_j$. Ví dụ, $Z = <B, C, D, B>$ là một dãy con của $X = <A, B, C, B, D, A, B>$ có dãy chỉ số tương ứng <2, 3, 5, 7>.

Cho hai dãy X và Y, ta nói rằng một dãy Z là một ***dãy con chung*** của X và Y nếu Z là một dãy con của cả X lẫn Y. Ví dụ, nếu $X = <A, B, C, B, D, A, B>$ và $Y = <B, D, C, A, B, A>$, dãy $<B, C, A>$ là một dãy con chung của cả X lẫn Y. Dãy $<B, C, A>$ không phải là dãy con chung *dài nhất* (longest common subsequence_LCS) của X và Y, bởi nó có chiều dài 3 và dãy (B, C, B, A), cũng là chung cho cả X lẫn Y, có chiều dài 4. Dãy $<B, C, B, A>$ là một LCS của X và Y, cũng như dãy $<B, D, A, B>$, bởi không có dãy con chung có chiều dài 5 hoặc lớn hơn.

Trong ***bài toán dãy con chung dài nhất***, ta có hai dãy $X = <x_1, x_2,..., x_m>$ *và* $Y = <y_1 , y_2,..., y_n >$ và muốn tìm ra một dãy con chung có chiều dài cực đại của X và Y. Đoạn này cho thấy bài toán LCS có thể được giải quyết hiệu quả bằng kỹ thuật lập trình động.

Mô tả đặc tính của một dãy con chung dài nhất

Một cách tiếp cận cưỡng bức để giải quyết bài toán LCS đó là điểm danh tất cả các dãy con của X và kiểm tra mỗi dãy con để xem nó cũng có phải là một dãy con của Y hay không, đồng thời theo dõi dãy con dài nhất đã tìm thấy. Mỗi dãy con của X tương ứng với một tập hợp con gồm các chỉ số $\{1, 2,..., m\}$ của X. Có 2^m dãy con của X, do đó cách tiếp cận này yêu cầu thời gian mũ, khiến nó không thực tiễn đối với các dãy dài.

Tuy nhiên, bài toán LCS có một tính chất cấu trúc con tối ưu, như định lý dưới đây đã nêu. Như sẽ thấy, lớp tự nhiên của các bài toán con tương ứng với các cặp "tiền tố" của hai dãy nhập liệu. Để chính xác, cho một dãy $X = <x_1, x_2,..., x_m >$, ta định nghĩa ***tiền tố*** thứ i của X, với $i = 0, 1,..., m$, vì $X_i = (x_1, x_2,..., x_i)$. Ví dụ, nếu $X = <A, B, C, B, D, A, B>$, thì $X_4 = <A, B, C, B>$ và X_0 là dãy trống.

Định lý 16.1 **(*Cấu trúc con tối ưu của một LCS*)**

Cho $X = <x_1, x_2,..., x_m>$ và $Y = <y_1, y_2,..., y_n>$ là các dãy, và cho $Z = <z_1, z_2,..., z_k>$ là một LCS bất kỳ của X và Y.

1. Nếu $x_m = y_n$, thì $z_k = x_m = y_n$ và Z_{k-1} là một LCS của X_{m-1} và Y_{n-1}.
2. Nếu $x_m \neq y_n$, thì $z_k \neq x_m$ hàm ý Z là một LCS của X_{m-1} và Y.
3. Nếu $x_m \neq y_n$, thì $z_k \neq y_n$ hàm ý Z là một LCS của X và Y_{n-1}.

Chứng minh (1) Nếu $z_k \neq x_m$, thì ta có thể chắp $x_m \neq y_n$ vào Z để có được một dãy con chung của X và Y có chiều dài $k + 1$, mâu thuẫn với giả thuyết cho rằng Z là một *dãy con chung dài nhất* của X và Y. Như vậy, ta phải có $z_k = x_m = y_n$. Giờ đây, tiền tố Z_{k-1} là một dãy con chung có chiều dài (k - 1) của X_{m-1} và Y_{n-1}. Ta muốn chứng tỏ nó là một LCS. Với mục tiêu là sự mâu thuẫn, giả sử có một dãy con chung W gồm X_{m-1} và Y_{n-1} có chiều dài lớn hơn k - 1. Như vậy, việc chắp $x_m = y_n$ vào W sẽ tạo ra một dãy con chung của X và Y có chiều dài lớn hơn k, là một sự mâu thuẫn.

(2) Nếu $z_k \neq x_m$, thì Z là một dãy con chung của X_{m-1} và Y. Nếu có một dãy con chung W gồm X_{m-1} và Y có chiều dài lớn hơn k, thì W cũng sẽ là một dãy con chung của X_n, và Y, mâu thuẫn với giả thiết cho rằng Z là một LCS của X và Y.

(3) Phần chứng minh đối xứng với (2).

Việc mô tả đặc tính của Định lý 16.1 cho thấy một LCS của hai dãy chứa trong nó một LCS gồm các tiền tố của hai dãy. Như vậy, bài toán LCS có một tính chất cấu trúc con tối ưu. Một giải pháp đệ quy cũng có tính chất các bài toán con phủ chồng, như ta sẽ thấy dưới đây.

Một giải pháp đệ quy cho các bài toán con

Định lý 16.1 hàm ý có một hoặc hai bài toán con để xem xét khi tìm một LCS của $X = <x_1, x_2,..., x_m>$ và $Y = <y_1, y_2,..., y_n>$. Nếu $x_m = y_n$ ta phải tìm một LCS của X_{m-1} và Y_{n-1}. Việc chắp $x_m = y_n$ vào LCS này cho ra một LCS của X và Y. Nếu $x_m \neq y_n$, thì ta phải giải quyết hai bài toán con: tìm một LCS của X_{m-1} và Y và tìm một LCS của X và Y_{n-1}. Cái nào trong hai LCS này dài hơn thì nó là một LCS của X và Y.

Ta có thể dễ dàng thấy tính chất các bài toán con phủ chồng trong bài toán LCS. Để tìm một LCS của X và Y, có thể ta cần phải tìm các LCS của X và Y_{n-1} và của X_{m-1} và Y. Nhưng mỗi bài toán con này đều có bài toán cháu tìm LCS của X_{m-1} và Y_{n-1}. Nhiều bài toán con khác chia sẻ các bài toán cháu.

Cũng như bài toán nhân xích ma trận, giải pháp đệ quy của ta đối với

bài toán LCS có liên quan đến việc thiết lập một phép truy toán cho mức hao phí của một giải pháp tối ưu. Ta hãy định nghĩa $c[i, j]$ là chiều dài của một LCS của các dãy X_i và Y_j. Nếu $i = 0$ hoặc $j = 0$, một trong các dãy sẽ có chiều dài 0, do đó LCS có chiều dài 0. Cấu trúc con tối ưu của bài toán LCS cho công thức đệ quy

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\,, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j\,, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j\,. \end{cases} \quad (16.5)$$

Tính toán chiều dài của một LCS

Dựa trên phương trình (16.5), ta có thể dễ dàng viết một thuật toán đệ quy thời gian mũ để tính toán chiều dài của một LCS của hai dãy. Tuy nhiên, do chỉ có $\Theta(mn)$ bài toán con riêng biệt, nên ta có thể dùng lập trình động để tính toán các giải pháp từ dưới lên.

Thủ tục LCS-LENGTH tiếp nhận hai dãy $X = \langle x_1, x_2, \ldots, x_m \rangle$ và $Y = \langle y_1, y_2, \ldots, y_n \rangle$ làm nhập liệu. Nó lưu trữ $c[i, j]$ giá trị trong một bảng $c[0\,.\,.\,m, 0\,.\,.\,n]$ có các khoản nhập được tính toán theo thứ tự chính là hàng. (Nghĩa là, hàng đầu tiên của c được điền từ trái qua phải, sau đó là hàng thứ hai, vân vân.) Nó cũng duy trì bảng $b[1..m, 1..n]$ để rút gọn việc kiến tạo một giải pháp tối ưu. Theo trực giác, $b[i, j]$ trỏ đến khoản nhập bảng tương ứng với giải pháp bài toán con tối ưu được chọn khi tính toán $c[i, j]$. Thủ tục trả về các bảng b và c; $c[m, n]$ chứa chiều dài của một LCS của X và Y.

```
LCS-LENGTH(X, Y)
m ← length[X]
n ← length[Y]
for i ← 1 to m
    do c[i, 0] ← 0
for j ← 0 to n
    do c[0, j] 4 ← 0
for i ← 1 to m
    do for j ← 1 to n
        do if x_i = y_j
            thì c[i,j] ← c[i - 1, j - 1]+1
                b[i j] ← "↖"
            else if c[i - 1, j] ≥ c[i, j - 1]
```

```
                         then  c[i,j] ← c[i - 1, j]
                               b[i j] ← "↑"
                         else  c[i, j] ← c[i, j - 1 ]
                               b[i, j] ← "←"
return c và b
```

Hình 16.3 nêu các bảng mà LCS-LENGTH tạo ra trên các dãy X = <A, B, C, B, D, A, B> và Y = <B, D, C, A, B, A>. Thời gian thực hiện của thủ tục là $O(mn)$, bởi mỗi khoản nhập bảng mất $O(1)$ thời gian để tính toán.

Kiến tạo một LCS

Bảng b mà LCS-LENGTH trả về có thể được dùng để nhanh chóng kiến tạo một LCS của $X = \langle x_1, x_2, ..., x_m \rangle$ và $Y = \langle y_1, y_2, ..., y_n \rangle$. Ta đơn giản bắt đầu tại $b[m, n]$ và rà qua bảng theo các mũi tên. Mỗi khi ta gặp một " ↖ " trong khoản nhập $b[i, j]$, nó hàm ý rằng $x_i = y_j$ là một thành phần của LCS. Phương pháp này gặp các thành phần của LCS theo thứ tự đảo ngược. Thủ tục đệ quy dưới đây in ra một LCS của X và Y trong thứ tự tiến tới đúng đắn. Lệnh gọi ban đầu là PRINT-LCS(b, X, *length*[X], *length*[Y]).

i \ j		0	1	2	3	4	5	6
	y_j		[illegible]	D	[illegible]	A	[illegible]	[illegible]
0	x_i	0	0	0	0	0	0	0
1	A	[illegible]	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	[illegible]	0	↖ 1	[illegible]	← 1	↑ 1	↖ 2	← 2
3	[illegible]	0	↑ 1	↑ 1	2	[illegible]	↑ 2	↑ 2
4	[illegible]	0	↖ 1	↑ 1	↑ 2	↑ 2	3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	[illegible]	↑ 3
6	[illegible]	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	[illegible]

Hình 16.3 Các bảng c và b mà LCS-LENGTH tính toán trên các dãy X = <A, B, C, B, D, A, B> và Y = <B, D, C, A, B, A>. Hình vuông trong hàng i và cột j chứa giá trị của $c[i, j]$ và mũi tên thích hợp cho giá trị của $b[i, j]$. Khoản nhập 4 trong $c[7, 6]$—góc dưới phải của bảng—là chiều dài của một LCS <B, C, B, A> của X và Y. Với $i, j > 0$, khoản nhập $c[i, j]$ chỉ tùy thuộc vào việc $x_i = y_j$ hay không và các giá trị trong các khoản nhập $c[i - 1, j]$, $c[i, j - 1]$, và $c[i - 1, j - 1]$, được tính toán trước $c[i, j]$. Để kiến tạo lại các thành phần của một LCS, bạn theo $b[i, j]$ mũi tên từ góc dưới phải; lộ trình được tô bóng. Mỗi " ↖ " trên lộ trình tương ứng với một khoản nhập (được nêu bật) qua đó $x_i = y_j$ là một phần tử của một LCS.

```
PRINT-LCS(b, X, i, j)
if i = 0 or j = 0
    then return
if b[i, j] = "↖"
    then PRINT-LCS(b, X, i - 1, j - 1)
         print x_i
else if b[i, j] = "↑"
    then PRINT-LCS(b, X, i - 1, j)
else PRINT-LCS(b, X, i, j - 1)
```

Với bảng b trong Hình 16.3, thủ tục này in *"BCBA."* Thủ tục mất một thời gian $O(m + n)$, bởi ít nhất một trong số i và j được giảm lượng trong mỗi giai đoạn của đệ quy.

Cải thiện mã

Sau khi phát triển một thuật toán, bạn thường thấy có thể cải thiện thời gian hoặc không gian mà nó sử dụng. Điều này đặc biệt đúng đối với các thuật toán lập trình động không phức tạp. Vài thay đổi có thể rút gọn mã và cải thiện các thừa số bất biến mà bằng không sẽ không có sự cải thiện tiệm cận nào trong khả năng thực hiện. Các thay đổi khác có thể mang lại các khoản tiết kiệm tiệm cận đáng kể về thời gian và không gian.

Ví dụ, có thể loại bỏ bảng b. Mỗi khoản nhập $c[i, j]$ chỉ tùy thuộc vào ba khoản nhập bảng c: $c[i - 1, j - 1]$, $c[i - 1, j]$, và $c[i, j - 1]$. Cho giá trị $c[i, j]$, ta có thể xác định trong $O(1)$ thời gian giá trị nào trong số ba giá trị này đã được dùng để tính toán $c[i, j]$, mà không kiểm tra bảng b. Như vậy, ta có thể liên tục lại một LCS trong $O(m + n)$ thời gian nhờ dùng một thủ tục tương tự như PRINT-LCS. (Bài tập 16.3-2 yêu cầu bạn cung cấp mã giả.) Tuy ta tiết kiệm $\Theta(mn)$ không gian bằng phương pháp này, song yêu cầu không gian phụ để tính toán một LCS không giảm theo tiệm cận, bởi ta vẫn cần $\Theta(mn)$ không gian cho bảng c.

Tuy nhiên, ta có thể rút gọn các yêu cầu không gian tiệm cận cho LCS-LENGTH, bởi nó chỉ cần hai hàng của bảng c vào một thời điểm: hàng đang được tính toán và hàng trước đó. (Thực vậy, ta có thể chỉ dùng nhiều hơn chút đỉnh so với không gian dành cho một hàng của c để tính toán chiều dài của một LCS. Xem Bài tập 16.3-4.) Kiểu cải thiện này làm việc nếu ta chỉ cần chiều dài của một LCS; nếu ta cần kiến tạo lại các thành phần của một LCS, bảng nhỏ hơn không lưu giữ đủ thông tin để rà lại các bước của chúng ta trong $O(m + n)$ thời gian.

Bài tập

16.3-1

Xác định một LCS của <1, 0, 0, 1, 0, 1, 0, 1> và <0, 1, 0, 1, 1, 0, 1, 1, 0>.

16.3-2

Nêu cách kiến tạo lại một LCS từ bảng *c* hoàn chỉnh và các dãy ban đầu $X = <x_1, x_2,..., x_m>$ và $Y = <y_1, y_2, ..., y_n>$ trong $O(m + n)$ thời gian, mà không dùng bảng *b*.

16.3-3

Nêu một phiên bản memo của LCS-LENGTH chạy trong $O(mn)$ thời gian.

16.3-4

Nêu cách tính toán chiều dài của một LCS chỉ dùng 2 min(m, n) khoản nhập trong bảng *c* cộng $O(1)$ không gian bổ sung. Sau đó, nêu cách thực hiện điều này bằng min(m, n) khoản nhập cộng $O(1)$ không gian bổ sung.

16.3-5

Nêu một thuật toán $O(n^2)$ thời gian để tìm dãy con tăng đơn điệu dài nhất của một dãy *n* con số.

16.3-6 *

Nêu một thuật toán $O(n \lg n)$ thời gian để tìm dãy con tăng đơn điệu dài nhất của một dãy *n* con số. (*Mách nước:* Nhận thấy thành phần cuối của một dãy con ứng viên có chiều dài *i* ít nhất cũng lớn bằng thành phần cuối của một dãy con ứng viên có chiều dài *i* - 1. Duy trì các dãy con ứng viên bằng cách nối kết chúng qua dãy nhập liệu.)

16.4 Phép tam giác phân đa giác tối ưu

Trong đoạn này, ta nghiên cứu bài toán tam giác phân tối ưu một đa giác lồi. Bất chấp dáng vẻ lồi ra ngoài của nó, ta sẽ thấy bài toán hình học này rất giống với phép nhân xích ma trận.

Một ***đa giác*** là một đường cong khép kín tuyến tính toàn khối trong mặt phẳng. Nghĩa là, nó là một đường cong kết thúc trên chính nó, hình thành bởi một dãy các đoạn thẳng, có tên là các ***cạnh*** của đa giác. Một

điểm ghép nối hai cạnh liên tiếp được gọi là một ***đỉnh*** của đa giác. Nếu đa giác là ***đơn,*** như ta thường mặc nhận, nó không tự giao nhau. Tập hợp các điểm trong mặt phẳng được bao bởi một đa giác đơn hình thành nên ***phần trong*** của đa giác, tập hợp các điểm trên chính đa giác hình thành nên ***biên*** [boundary] của nó, và tập hợp các điểm bao quanh đa giác hình thành ***phần ngoài*** của nó. Một đa giác đơn là ***lồi*** nếu, cho hai điểm bất kỳ trên biên hoặc trong phần trong của nó, tất cả các điểm trên đoạn thẳng vẽ giữa chúng hàm chứa trong biên hoặc phần trong của đa giác.

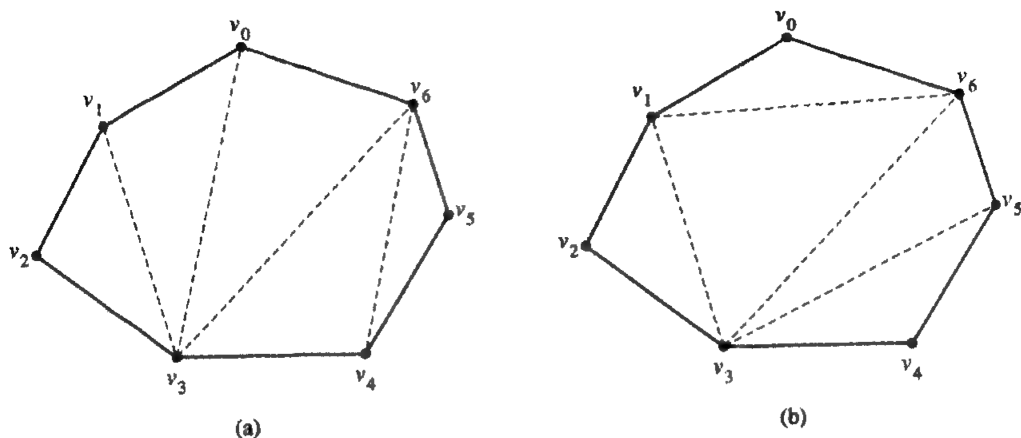
Để biểu diễn một đa giác lồi ta có thể liệt kê các đỉnh của nó theo thứ tự ngược chiều kim đồng hồ. Nghĩa là, nếu $P = \langle v_0, v_1, \ldots, v_{n-1} \rangle$ là một đa giác lồi, nó có n cạnh $\overline{v_0v_1}, \overline{v_1 v_2}, \ldots, \overline{v_{n-1}, v_n}$, ở đó ta diễn dịch v_n là v_o. (Nói chung, ta sẽ mặc nhận số học trên các chỉ số đỉnh được lấy modulo số lượng các đỉnh.)

Cho hai đỉnh không liền kề v_i và v_j, đoạn $\overline{v_iv_j}$ là một ***dây cung*** của đa giác. Một dây cung $\overline{v_i v_j}$ chia đa giác thành hai đa giác: $\langle v_i, v_{i+1}, \ldots, v_j \rangle$ và $\langle v_j, v_{j+1}, \ldots, v_i \rangle$. Một ***phép tam giác phân*** [triangulation] của một đa giác là một tập hợp T gồm các dây cung của đa giác chia đa giác thành các ***tam giác*** rời nhau (các đa giác có 3 cạnh). Hình 16.4 nêu hai cách tam giác phân một đa giác có 7 cạnh. Trong một phép tam giác phân, không có dây cung giao nhau (ngoại trừ tại các điểm cuối) và tập hợp T các dây cung là cực đại: mọi dây cung không nằm trong T sẽ giao nhau với một dây cung nào đó trong T. Các cạnh của các tam giác do phép tam giác phân tạo ra sẽ là dây cung trong phép tam giác phân hoặc các cạnh của đa giác. Mọi phép tam giác phân của một đa giác lồi n-đỉnh có n - 3 dây cung và chia đa giác thành n - 2 tam giác.

Trong ***bài toán tam giác phân (đa giác) tối ưu,*** ta có một đa giác lồi $P = \langle v_0, v_1, \ldots, v_{n-1} \rangle$ và một hàm trọng số w được định nghĩa trên các tam giác được hình thành bởi các cạnh và các dây cung của P. Bài toán đó là tìm một phép tam giác phân giảm thiểu tổng của các trọng số của các tam giác trong phép tam giác phân. Một hàm trọng số [weight fuction] trên các tam giác xuất hiện trong tâm trí một cách tự nhiên đó là

$$w(\triangle v_iv_jv_k) = |v_iv_j| + |v_jv_k| + |v_kv_i| ,$$

ở đó $|v_iv_j|$ (là khoảng cách euclid từ v_i các v_j. Thuật toán mà ta sẽ phát triển làm việc với một chọn lựa hàm trọng số tùy ý.



Hình 16.4 Hai cách tam giác phân một đa giác lồi. Mọi phép tam giác phân của đa giác có 7 cạnh này có 7 - 3 = 4 dây cung và chia đa giác thành 7 - 2 = 5 tam giác.

Tương ứng phép ngoặc đơn

Có một tương ứng đáng ngạc nhiên giữa phép tam giác phân của một đa giác và phép ngoặc đơn của một biểu thức như một tích xích ma trận. Sự tương ứng này được giải thích tốt nhất bằng các cây.

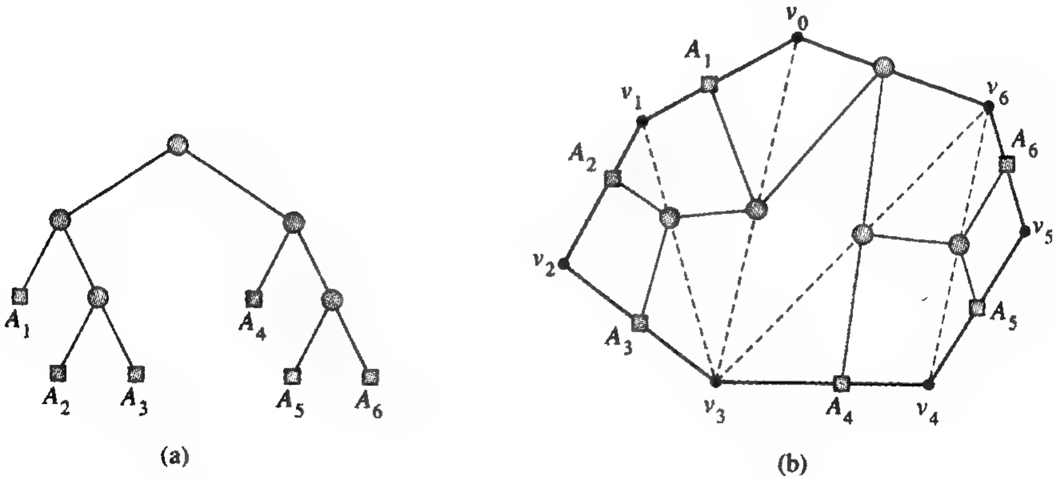
Một phép ngoặc đơn đầy đủ của một biểu thức tương ứng với một cây nhị phân đầy đủ, đôi lúc còn gọi là ***cây phân ngữ*** [parse tree] của biểu thức. Hình 16.5(a) nêu một cây phân ngữ cho tích xích ma trận đã ngoặc đơn

$$((A_1A_2A_3))(A_4(A_5A_6))) \ . \qquad (16.6)$$

Mỗi lá của một cây phân ngữ được gán nhãn bằng một trong các thành phần nguyên tử (các ma trận) trong biểu thức. Nếu gốc của một cây con của cây phân ngữ có một cây con trái biểu thị cho một biểu thức E_l và một cây con phải biểu thị cho một biểu thức E_r, thì bản thân cây con đó biểu thị cho biểu thức $(E_l E_r)$. Có một sự tương ứng một-một giữa các cây phân ngữ và các biểu thức được ngoặc đơn đầy đủ trên n thành phần nguyên tử.

Phép tam giác phân của một đa giác lồi $<v_0, v_1, ..., v_{n-1}>$ cũng có thể được biểu diễn bởi một cây phân ngữ. Hình 16.5(**b**) nêu cây phân ngữ của phép tam giác phân của đa giác trong Hình 16.4(**a**). Các nút trong của cây phân ngữ là các dây cung của phép tam giác phân cộng với cạnh $\overline{v_0v_6}$, là gốc. Các lá là các cạnh khác của đa giác. Gốc $\overline{v_0v_6}$ là một cạnh của tam giác $\triangle v_0v_3v_6$. Tam giác này xác định các con của gốc: một là dây cung $\overline{v_0v_3}$, và một là dây cung $\overline{v_3v_6}$. Lưu ý, tam giác này chia đa giác ban đầu thành ba phần: chính tam giác $\triangle v_0v_3v_6$, đa giác $<v_0, v_1, ...,$

v_3>, và đa giác <v_3, v_4,..., v_6>. Hơn nữa, hai đa giác con được hình thành trọn vẹn bởi các cạnh của đa giác ban đầu, ngoại trừ các gốc của chúng, là các dây cung $\overline{v_0v_3}$ và $\overline{v_3v_6}$.



Hình 16.5 Các cây phân ngữ. **(a)** Cây phân ngữ của tích đã ngoặc đơn ((A_1 (A_2A_3))(A_4(A_5A_6))) và của phép tam giác phân của đa giác có 7 cạnh trong Hình 16.4(a). **(b)** Phép tam giác phân của đa giác có cây phân ngữ được phủ chồng. Mỗi ma trận A; tương ứng với cạnh $\overline{v_{i-1},v_i}$ với i = 1, 2,..., 6.

Theo dạng đệ quy, đa giác <v_0, v_1,..., v_3> chứa cây con trái của gốc của cây phân ngữ, và đa giác <v_3, v_4,..., v_6> chứa cây con phải.

Do đó, nói chung, một phép tam giác phân của một đa giác có n cạnh tương ứng với một cây phân ngữ có n - 1 lá. Bằng một tiến trình nghịch đảo, ta có thể tạo ra một phép tam giác phân từ một cây phân ngữ đã cho. Có một sự tương ứng một-một giữa các cây phân ngữ và các phép tam giác phân.

Do một tích được ngoặc đơn đầy đủ gồm n ma trận tương ứng với một cây phân ngữ có n lá, nên nó cũng tương ứng với một phép tam giác phân của một đa giác (n + 1) đỉnh. Các Hình 16.5(a) và (b) minh họa sự tương ứng này. Mỗi ma trận A_i trong một tích A_1 A_2 ... A_n tương ứng với một cạnh v_{i-1} v_i của một đa giác (n + 1) đỉnh. Một dây cung $v_{i-1}v_j$, ở đó i < j, tương ứng với một ma trận $A_{i+1..j}$ đã tính toán trong khi đánh giá tích.

Thực vậy, bài toán nhân xích ma trận là một trường hợp đặc biệt về bài toán tam giác phân tối ưu. Nghĩa là, mọi trường hợp của phép nhân xích ma trận có thể được tính như một bài toán tam giác phân tối ưu. Cho một tích xích ma trận A_1A_2 ... A_n, ta định nghĩa một đa giác lồi (n + 1) đỉnh P = <v_0, v_1,..., v_n>. Nếu ma trận A_i có các chiều p_{i-1} x p_i với i = 1, 2,...,

n, ta định nghĩa hàm trọng số của phép tam giác phân là

$w(\triangle v_i v_j v_k) = p_i p_j p_k$

Một phép tam giác phân tối ưu của P đối với hàm trọng số này sẽ cho ra cây phân ngữ của một phép ngoặc đơn tối ưu của $A_1 A_2 \ldots A_n$.

Mặc dù phép nghịch đảo không true—bài toán tam giác phân tối ưu không phải là một trường hợp đặc biệt của bài toán nhân xích ma trận—song hóa ra, với một vài sửa đổi nhỏ, mã MATRIX-CHAIN-ORDER của chúng ta trong Đoạn 16.1 sẽ giải quyết bài toán tam giác phân tối ưu trên một đa giác $(n + 1)$ đỉnh. Ta đơn giản thay dãy $<p_0, p_1, \ldots, p_n>$ của các chiều ma trận bằng dãy $<v_0, v_1, \ldots, v_n>$ của các đỉnh, thay đổi các tham chiếu đến p thành các tham chiếu đến v, và thay đổi dòng 9 thành:

9 **do** $q \leftarrow m[i, k] + m[k + 1, j] + w(\triangle v_{i-1} v_k v_j)$

Sau khi chạy thuật toán, thành phần $m[1, n]$ chứa trọng số của một phép tam giác phân tối ưu. Ta hãy xem tại sao lại như vậy.

Cấu trúc con của một phép tam giác phân tối ưu

Xét một phép tam giác phân tối ưu T của một đa giác $(n + 1)$-đỉnh $P = <v_0, v_1, \ldots, v_n>$ bao gồm tam giác $\triangle v_0 v_k v_n$, với vài k, ở đó $1 \leq k \leq n - 1$. Trọng số của T chính là tổng của các trọng số của $\triangle v_0 v_k v_n$ và các tam giác trong phép tam giác phân của hai đa giác con $<v_0, v_1, \ldots, v_k>$ và $<v_k, v_{k+1}, \ldots, v_n>$. Do đó, các phép tam giác phân của các đa giác con được xác định bởi T phải là tối ưu, bởi một phép tam giác phân có trọng số ít hơn của một trong hai đa giác con sẽ mâu thuẫn với tính cực tiểu của trọng số của T.

Một giải pháp đệ quy

Hệt như đã định nghĩa $m[i, j]$ là mức hao phí cực tiểu của việc tính toán tích con xích ma trận $A_i A_{i+1} \ldots A_j$, ta hãy định nghĩa $t[i, j]$, với $1 \leq i < j \leq n$, là trọng số của một phép tam giác phân tối ưu của đa giác $<v_{i-1}, v_i, \ldots, v_j>$. Để tiện dụng, ta xét một đa giác suy biến $<v_{i-1}, v_i>$ để có trọng số 0. Trọng số của một phép tam giác phân tối ưu của đa giác P được sinh ra bởi $t[1, n]$.

Bước kế tiếp của chúng ta đó là định nghĩa $t[i, j]$ theo đệ quy. Cơ sở là trường hợp suy biến của một đa giác 2-đỉnh: $t[i, i] = 0$ với $i = 1, 2, \ldots, n$. Khi $j - i \geq 1$, ta có một đa giác $<v_{i-1}, v_i, \ldots, v_j>$ với ít nhất 3 đỉnh. Ta muốn giảm thiểu trên tất cả các đỉnh v_k, với $k = i, i + 1, \ldots, j - 1$, trọng số của $\triangle v_{i-1} v_k v_j$ cộng với các trọng số của các phép tam giác phân tối ưu của các đa giác $<v_{i-1}, v_i, \ldots, v_k>$ và $<v_k, v_{k+1}, \ldots, v_j>$. Như vậy, phép lập đệ quy là

$$t[i,j] = \begin{cases} 0 & \text{nếu } i = j\,, \\ \min\limits_{i \le k \le j \le j-1} \{t[i,k] + t[k+1,j] + w(\triangle\ v_{i-1}v_kv_j)\} & \text{nếu } i < j\,. \end{cases} \quad (16.7)$$

So sánh phép truy toán này với phép truy toán (16.2) mà ta đã phát triển cho số cực tiểu $m[i, j]$ các phép nhân vô hướng cần có để tính toán $A_iA_{i+1}...A_j$. Ngoại trừ hàm trọng số, các phép truy toán là đồng nhất, và như vậy, với vài thay đổi nhỏ đối với mã đã nêu trên đây, thủ tục MATRIX-CHAIN-ORDER có thể tính toán trọng số của một phép tam giác phân tối ưu. Cũng như MATRIX-CHAIN-ORDER, thủ tục phép tam giác phân tối ưu chạy trong thời gian $\Theta(n^3)$ và sử dụng $\Theta(n^2)$ không gian.

Bài tập

16.4-1

Chứng minh mọi phép tam giác phân của một đa giác lồi n-đỉnh có $n - 3$ dây cung và chia đa giác thành $n - 2$ tam giác.

16.4-2

Giáo sư Guinevere gợi ý rằng có thể có một thuật toán nhanh hơn để giải quyết bài toán tam giác phân tối ưu cho trường hợp đặc biệt ở đó trọng số của một tam giác là diện tích của nó. Trực giác của giáo sư có chính xác không?

16.4-3

Giả sử một hàm trọng số w được định nghĩa trên các dây cung của một phép tam giác phân thay vì trên các tam giác. Như vậy trọng số của một phép tam giác phân đối với w là tổng các trọng số của các dây cung trong phép tam giác phân. Chứng tỏ bài toán tam giác phân tối ưu có các dây cung đã gia trọng [weighted] chẳng qua là một trường hợp đặc biệt của bài toán tam giác phân tối ưu có các tam giác đã gia trọng.

16.4-4

Tìm một phép tam giác phân tối ưu của một bát giác đều có các cạnh có chiều dài đơn vị. Dùng hàm trọng số

$$w(\triangle v_iv_jv_k) = |v_i\, v_j| + |v_j\, v_k| + |v_k\, v_i|$$

ở đó $|v_i\, v_j|$ là khoảng cách euclid từ v_i đến v_j. (Một đa giác đều là một đa giác có các cạnh bằng nhau và các góc trong bằng nhau.)

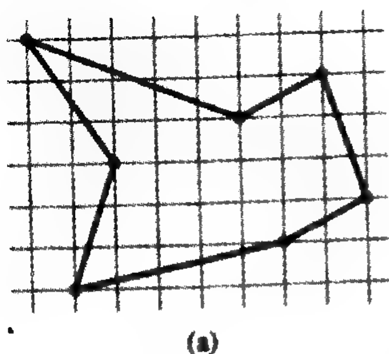
Các Bài Toán

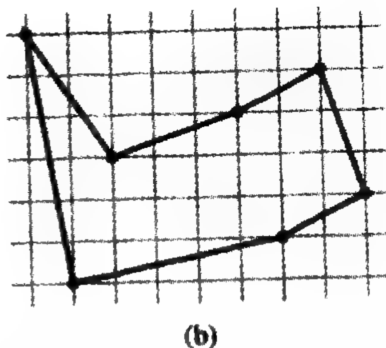
16-1 Bài toán nhân viên bán hàng di chuyển euclid bitonic

Bài toán nhân viên bán hàng di chuyển euclid [euclidean traveling-salesman problem] là bài toán xác định chuyến đi khép kín ngắn nhất nối một tập hợp *n* điểm đã cho trong mặt phẳng. Hình 16.6(a) có nêu giải pháp cho một bài toán 7 điểm. Bài toán chung là đầy đủ NP [NP-complete], và như vậy giải pháp của nó được tin là yêu cầu nhiều hơn thời gian đa thức (xem Chương 36).

J. L. Bentley đã gợi ý rằng ta rút gọn bài toán bằng cách hạn chế sự chú ý vào các ***chuyến đi bitonic,*** nghĩa là, các chuyến đi bắt đầu tại điểm mút trái, đi hoàn toàn từ trái qua phải đến điểm mút phải, rồi đi hoàn toàn từ phải qua trái đến điểm bắt đầu. Hình 16.6(b) nêu chuyến đi bitonic ngắn nhất của cùng 7 điểm. Trong trường hợp này, một thuật toán thời gian đa thức là khả dĩ.

Mô tả một thuật toán $O(n^2)$ thời gian để xác định một chuyến đi bitonic tối ưu. Bạn có thể mặc nhận rằng không có hai điểm có cùng tọa độ *x*. (*Mách nước*: Quét từ trái qua phải, duy trì các khả năng tối ưu cho hai phần của chuyến đi.)





(a) **(b)**

Hình 16.6 Bảy điểm trong mặt phẳng, được nêu trên một khung kẻ ô đơn vị. **(a)** Chuyến đi khép kín ngắn nhất, có chiều dài 24.88....Chuyến đi này không phải là bitonic. **(b)** Chuyến đi bitonic ngắn nhất cho cùng tập hợp các điểm. Chiều dài của nó là 25.58....

16-2 In gọn

Xét bài toán in gọn [neatly printing] một đoạn trên một máy in. Văn bản nhập liệu là một dãy *n* từ có các chiều dài $l_1, l_2, \ldots, l_n$, được đo bằng ký tự. Ta muốn in đoạn này gọn gàng trên một số dòng lưu giữ tối đa *M* ký tự cho mỗi dòng. Quy chuẩn về "sự gọn gàng" của chúng ta là như

sau. Nếu một dòng đã cho chứa các từ *i* đến *j* và ta chứa chính xác một không gian giữa các từ, lượng ký tự dấu cách phụ trội tại cuối dòng đó là $M - j + i - \sum_{k=i}^{j} l_k$. Ta muốn giảm thiểu tổng, trên tất cả các dòng ngoại trừ dòng cuối, các khối lập phương của các số ký tự dấu cách phụ trội tại cuối các dòng. Nêu một thuật toán lập trình động để in một đoạn *n* từ gọn gàng trên một máy in. Phân tích thời gian thực hiện và các yêu cầu không gian của thuật toán.

16-3 Hiệu chỉnh khoảng cách

Khi một trạm cuối "thông minh" cập nhật một dòng văn bản, thay một chuỗi "nguồn" hiện có $x[1..m]$ với một chuỗi "đích" mới $y[1..n]$, ta có vài cách để có thể thay đổi. Với một ký tự đơn lẻ của chuỗi nguồn, ta có thể xóa, thay bằng một ký tự khác, hoặc chép sang chuỗi đích; có thể chèn các ký tự; hoặc có thể hoán đổi ("xoay") hai ký tự kề nhau của chuỗi nguồn trong khi đang được chép sang chuỗi đích. Sau khi xảy ra tất cả các phép toán khác, có thể xóa nguyên một hậu tố của chuỗi nguồn, một phép toán có tên "triệt đến cuối dòng."

Để lấy ví dụ, một cách để biến đổi chuỗi nguồn algorithm thành chuỗi đích altruistic đó là sử dụng dãy phép toán dưới đây.

Phép toán	Chuỗi đích	Chuỗi nguồn
chép a	a	lgorithm
chép l	al	gorithm
thay g bằng t	alt	orithm
xóa o	alt	rithm
chép r	altr	ithm
chèn a	altru	ithm
chèn i	altrui	ithm
chèn s	altruis	ithm
xoay it thành ti	altruisti	hm
chèn c	altruistic	hm
triệt hm	altruistic	

Có nhiều dãy phép toán khác hoàn thành cùng kết quả.

Mỗi phép toán xóa, thay, chép, chèn, xoay, và triệt [kill] đều có một mức hao phí kết hợp. (Có lẽ, mức hao phí để thay một ký tự sẽ nhỏ hơn các mức hao phí phối hợp của phép xóa và phép chèn; bằng không, phép toán thay sẽ không được dùng.) Mức hao phí của một dãy các phép toán biến đổi đã cho là tổng các mức hao phí của các phép toán

riêng lẻ trong dãy. Với dãy trên đây, mức hao phí để chuyển đổi thuật toán thành **altruistic** là

(3 - mức hao phí(chép)) + mức hao phí(thay) + mức hao phí(xóa) + (3 - mức hao phí(chèn))

+ mức hao phí(xoay) + mức hao phí(triệt) .

Cho hai dãy $x[1..m]$ và $y[1..n]$ và một tập hợp các mức hao phí phép toán đã cho, ***khoảng cách hiệu chỉnh*** từ x đến y là mức hao phí của dãy biến đổi ít tốn kém nhất để chuyển đổi x thành y. Mô tả một thuật toán lập trình động để tìm khoảng cách hiệu chỉnh từ $x[1..m]$ đến $y[1..n]$ và in một dãy biến đổi tối ưu. Phân tích thời gian thực hiện và các yêu cầu không gian của thuật toán.

16-4 Hoạch định một buổi tiệc công ty

Giáo sư McKenzie đang tư vấn cho tổng giám đốc của công ty A.-B. Corporation, công ty này đang hoạch định một buổi tiệc công ty. Công ty có một cấu trúc phân cấp; nghĩa là, quan hệ giám sát viên hình thành một cây có gốc tại tổng giám đốc. Phòng nhân sự đã xếp hạng cho mỗi nhân viên theo cấp bậc khả năng ăn uống vui vẻ, là một số thực. Để buổi tiệc trở nên vui vẻ với tất cả những người tham dự, tổng giám đốc không muốn cả hai một nhân viên và giám sát viên trực tiếp của họ cùng tham dự.

a. Mô tả một thuật toán hình thành danh sách khách mời. Mục tiêu sẽ là tối đa hóa tổng các cấp bậc khả năng ăn uống vui vẻ của các khách mời. Phân tích thời gian thực hiện của thuật toán.

b. Làm sao để giáo sư có thể bảo đảm vị tổng giám đốc được mời đến buổi tiệc riêng của mình?

16-5 Thuật toán Viterbi

Ta có thể dùng lập trình động trên một đồ thị có hướng $G = (V, E)$ để nhận dạng tiếng nói. Mỗi cạnh $(u, v) \in E$ được gán nhãn bằng một âm thanh $\sigma(u, v)$ từ một tập hợp hữu hạn Σ âm thanh. Đồ thị gán nhãn là một mô hình hình thức của một người đang nói một ngôn ngữ hạn chế. Mỗi lộ trình trong đồ thị bắt đầu từ một đỉnh đặc biệt $v_0 \in V$ tương ứng với một dãy khả dĩ các âm thanh mà mô hình tạo ra. Nhãn của một lộ trình có hướng được định nghĩa là sự ghép nối các nhãn của các cạnh trên lộ trình đó.

a. Mô tả một thuật toán hiệu quả mà, căn cứ vào một đồ thị G có cạnh được gán nhãn với đỉnh đặc biệt v_0 và một dãy $s = (\sigma_1, \sigma_2, \ldots, \sigma_k)$ các ký tự từ Σ, sẽ trả về một lộ trình trong G bắt đầu tại v_0 và có s làm nhãn của nó, nếu có một lộ trình như vậy. Bằng không, thuật toán sẽ trả về

NO-SUCH-PATH. Phân tích thời gian thực hiện của thuật toán. (*Mách nước:* Bạn có thể thấy các khái niệm trong Chương 23 là hữu ích.)

Giờ đây, giả sử mọi cạnh $(u, v) \in E$ cũng có một xác suất không âm [nonnegative] kết hợp $p(u, v)$ của việc băng ngang cạnh (u, v) từ đỉnh u và tạo ra âm thanh tương ứng. Tổng các xác suất của các cạnh rời một đỉnh bất kỳ sẽ bằng 1. Xác suất của một lộ trình được định nghĩa là tích các xác suất của các cạnh của nó. Ta có thể xem xác suất của một lộ trình bắt đầu tại v_0 là xác suất mà một "tầng ngẫu nhiên" ["random walk"] bắt đầu tại v_0sẽ theo lộ trình đã chỉ định, ở đó xét theo xác suất việc chọn lựa sẽ lấy cạnh nào tại một đỉnh u được thực hiện theo các xác suất của các cạnh sẵn có rời u.

b. Mở rộng đáp án đến phần (a) sao cho nếu một lộ trình được trả về, thì nó là một *lộ trình khả dĩ nhất* [most probable] bắt đầu tại v_0 và có nhãn s. Phân tích thời gian thực hiện của thuật toán.

Ghi chú Chương

R. Bellman đã bắt đầu nghiên cứu có hệ thống về lập trình động vào năm 1955..Từ "lập trình," cả ở đây lẫn trong lập trình tuyến tính, ám chỉ việc dùng một phương pháp giải quyết theo bảng biểu. Mặc dù các kỹ thuật tối ưu hóa liên kết với các thành phần của lập trình động đã nêu trên đây, Bellman đã cung cấp cho diện tích một cơ sở toán học vững chắc [21].

Hu và Shing [106] cung cấp một thuật toán $O(n \lg n)$ thời gian cho bài toán nhân xích ma trận. Họ cũng chứng minh sự tương ứng giữa bài toán tam giác phân đa giác tối ưu và bài toán nhân xích ma trận.

Thuật toán $O(mn)$ thời gian của bài toán dãy con chung dài nhất dường như là một thuật toán dân gian. Knuth [43] đã đặt câu hỏi có hay không các thuật toán bình phương con [subquadratic algorithm] cho bài toán LCS. Masek và Paterson [143] đã trả lời khẳng định đối với câu hỏi này bằng cách cung cấp một thuật toán chạy trong $O(mn/ \lg n)$ thời gian, ở đó $n \leq m$ và các dãy được rút ra từ một tập hợp có kích cỡ giới hạn. Với trường hợp đặc biệt ở đó không có thành phần nào xuất hiện nhiều hơn một lần trong một dãy nhập liệu, Szymanski (184] cho thấy bài toán có thể được giải quyết trong $O((n + m) \lg(n + m))$ thời gian. Nhiều kết quả này đã mở rộng ra bài toán tính toán các khoảng cách hiệu chỉnh chuỗi (Bài toán 16-3).

17 Các Thuật Toán Tham

Các thuật toán cho các bài toán tối ưu hóa thường trải qua một dãy các bước, với một tập hợp các chọn lựa tại mỗi bước. Với nhiều bài toán tối ưu hóa, quả là quá thừa khi dùng lập trình động để xác định các chọn lựa tốt nhất; thay vì thế, ta có thể dùng các thuật toán đơn giản và hiệu quả hơn. Trước nút, ***thuật toán tham*** [greedy algorithm] luôn thực hiện sự chọn lựa có vẻ tốt nhất. Nghĩa là, nó thực hiện một sự chọn lựa tối ưu cục bộ với hy vọng chọn lựa này sẽ dẫn đến một giải pháp tối ưu toàn cục. Chương này khảo sát các bài toán tối ưu mà các thuật toán tham có thể giải quyết.

Tuy không phải lúc nào các thuật toán tham cũng cho ra các giải pháp tối ưu, song đa phần là vậy. Trước tiên, Đoạn 17.1 sẽ xem xét một bài toán tuy đơn giản song không phải tầm thường, bài toán lựa chọn hoạt động, mà một thuật toán tham có thể tính toán một giải pháp hiệu quả. Kế đó, Đoạn 17.2 ôn lại vài thành phần cơ bản của cách tiếp cận tham. Đoạn 17.3 trình bày một ứng dụng quan trọng về các kỹ thuật tham: thiết kế các mã nén dữ liệu (Huffman). Trong Đoạn 17.4, ta nghiên cứu vài lý thuyết làm cơ sở cho các cấu trúc tổ hợp có tên "các tạng ma trận" [matroids] mà một thuật toán tham luôn tạo ra giải pháp tối ưu. Cuối cùng, Đoạn 17.5 minh họa ứng dụng của các tạng ma trận dùng bài toán lên lịch các công việc thời gian đơn vị có các hạn chót và các hình phạt.

Phương pháp tham khá mạnh và làm việc tốt với nhiều bài toán. Các chương sau sẽ trình này nhiều thuật toán có thể được xem như các ứng dụng về phương pháp tham, kể cả các thuật toán cây tỏa nhánh cực tiểu (Chương 24), thuật toán của Dijkstra cho các lộ trình ngắn nhất từ một nguồn đơn lẻ (Chương 25), và heuristic phủ tập hợp tham [greedy set-covering heuristic] của Chvatal (Chương 37). Các cây tỏa nhánh cực tiểu hình thành một ví dụ cổ điển về phương pháp tham. Tuy chương này và Chương 24 có thể đọc độc lập với nhau, song bạn sẽ thấy hữu ích nếu như đọc chung với nhau.

17.1 Một bài toán lựa chọn hoạt động

Ví dụ đầu tiên của chúng ta đó là bài toán lên lịch một tài nguyên

giữa vài hoạt động cạnh tranh. Ta thấy rằng thuật toán tham sẽ cung cấp một phương pháp đơn giản và lịch lãm để chọn một tập hợp có kích cỡ cực đại các hoạt động tương thích lẫn nhau.

Giả sử ta có một tập hợp $S = \{1, 2,..., n\}$ gồm *n* ***hoạt động*** đã đề xuất muốn sử dụng một tài nguyên, như một giảng đường chẳng hạn, mà mỗi lần chỉ có thể được dùng bởi một hoạt động. Mỗi hoạt động *i* có một ***thời gian bắt đầu*** s_i và một ***thời gian kết thúc*** f_i ở đó $s_i \leq f_i$. Nếu được chọn, hoạt động *i* diễn ra trong quãng thời gian nửa mở $[s_i, f_i)$. Các hoạt động *i* và *j* là ***tương thích*** nếu các quãng $[s_i, f_i)$ và $[s_j, f_j)$ không phủ chồng (tức, *i* và *j* tương thích nếu $s_i \geq f_j$ hoặc $s_j \geq f_i$). ***Bài toán lựa chọn hoạt động*** sẽ chọn một tập hợp có kích cỡ cực đại các hoạt động tương thích lẫn nhau.

Mã giả dưới đây cho ta một thuật toán tham của bài toán lựa chọn hoạt động. Ta mặc nhận rằng các hoạt động nhập liệu được sắp xếp thứ tự theo thời gian kết thúc tăng:

$$f_1 \leq f_2 \leq ... \leq f_n, \qquad (17.1)$$

Nếu không, ta có thể sắp xếp chúng theo thứ tự này trong thời gian $O(n\lg n)$, phá vỡ các ràng buộc một cách tùy ý. Mã giả mặc nhận các nhập liệu *s* và *f* được biểu thị dưới dạng các mảng.

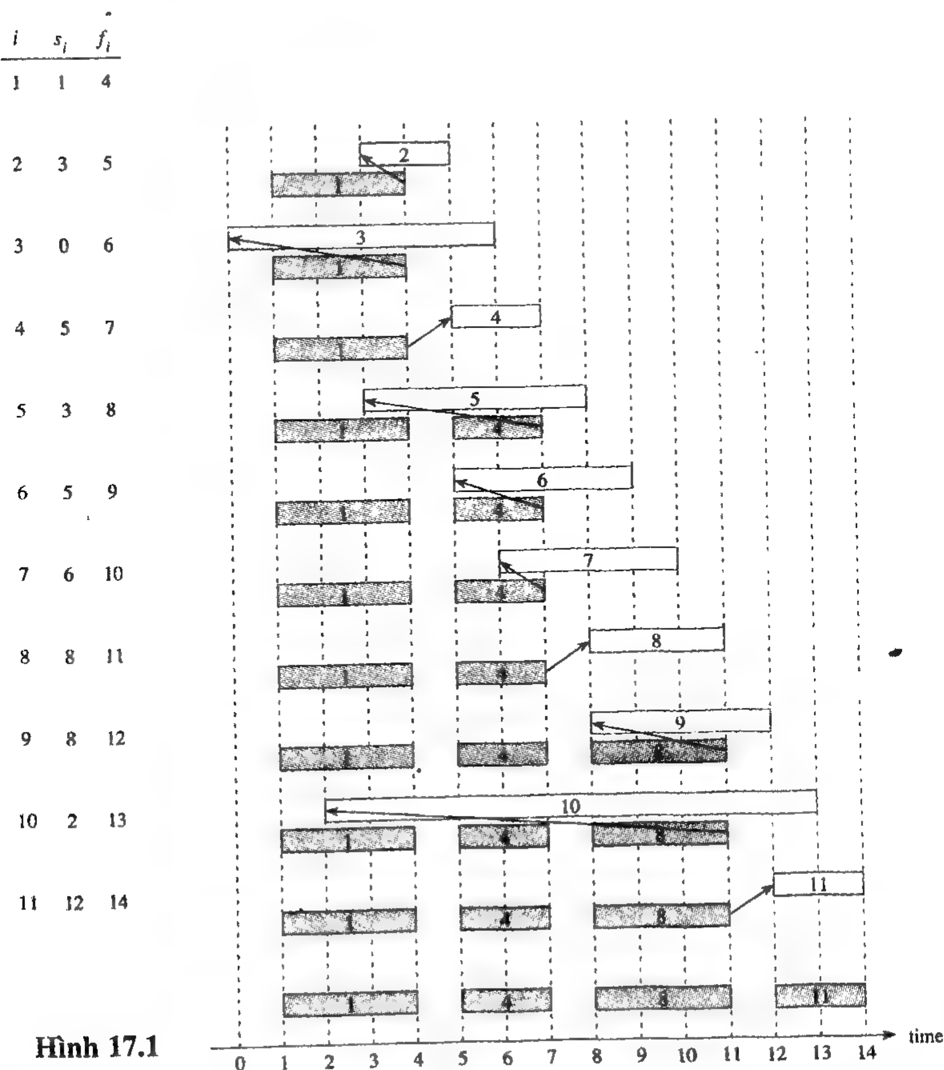
```
GREEDY-ACTIVITY-SELECTOR(s, f)
1 n ← length[s]
2 A ← {1}
3 j ← 1
4 for i ← 2 to n
5       do if s_i ≥ f_j
6               then A ← A ∪ {i}
7                    j ← i
8 return A
```

Phép toán của thuật toán được nêu trong Hình 17.1. Tập hợp *A* thu thập các hoạt động đã chọn. Biến *j* chỉ định phần bổ sung mới nhất vào *A*. Do các hoạt động được xem theo thứ tự của thời gian kết thúc không giảm, f_j luôn là thời gian kết thúc cực đại của bất kỳ hoạt động nào trong *A*. Nghĩa là,

$$f_j = \max\{f_k : k \in A\}. \qquad (17.2)$$

Các dòng 2-3 lựa hoạt động 1, khởi tạo *A* để chỉ chứa hoạt động này, và khởi tạo *j* cho hoạt động này. Các dòng 4-7 lần lượt xét từng hoạt động *i* và cộng *i* vào *A* nếu nó tương thích với tất cả các hoạt động đã lựa trước đó. Để xem hoạt động *i* có tương thích với mọi hoạt động hiện

ở trong A hay không, ta chỉ cần dùng phương trình (17.2) để kiểm tra (dòng 5) thời gian bắt đầu của nó s_i không sớm hơn thời gian kết thúc f_j của hoạt động mới được bổ sung vào A. Nếu hoạt động i tương thích, thì các dòng 6-7 sẽ cộng nó vào A và cập nhật j. Thủ tục GREEDY-ACTIVITY-SELECTOR khá hiệu quả. Nó có thể lên lịch một tập hợp S n hoạt động trong $\Theta(n)$ thời gian, giả định các hoạt động đã được sắp xếp từ đầu theo các thời gian kết thúc của chúng.



Hình 17.1 Phép toán của GREEDY-ACTIVITY-SELECTOR trên 11 hoạt động được cho bên trái. Mỗi hàng trong hình tương ứng với một lần lặp lại của vòng lặp **for** trong các dòng 4-7. Ta tô bóng các hoạt động đã được chọn nằm trong tập hợp A, và tô trắng hoạt động i đang được xét. Nếu thời gian bắt đầu s_i của hoạt động i xảy ra trước thời gian kết thúc f_j của hoạt động mới được chọn j (mũi tên giữa chúng trỏ trái), nó bị loại. Bằng không (mũi tên trỏ trực tiếp lên hoặc sang phải), nó được chấp nhận và đưa vào tập hợp A.

Hoạt động được GREEDY-ACTIVITY-SELECTOR lựa kế tiếp luôn là hoạt động có thời gian kết thúc sớm nhất có thể lên lịch một cách hợp pháp. Như vậy, hoạt động được lựa là một chọn lựa "tham" theo nghĩa là, theo trực giác, nó để lại càng nhiều cơ hội càng tốt cho các hoạt động còn lại được lên lịch. Nghĩa là, chọn lựa tham là chọn lựa tối đa hóa lượng thời gian không lên lịch còn lại.

Chứng minh thuật toán tham là đúng

Các thuật toán tham không luôn tạo ra các giải pháp tối ưu. Tuy nhiên, GREEDY-ACTIVITY-SELECTOR luôn tìm một giải pháp tối ưu cho một trường hợp của bài toán lựa chọn hoạt động.

Định lý 17.1

Thuật toán GREEDY-ACTIVITY-SELECTOR tạo ra các giải pháp có kích cỡ cực đại cho bài toán lựa chọn hoạt động.

Chứng minh Cho $S = \{1, 2,..., n\}$ là tập hợp của các hoạt động để lên lịch. Bởi ta đang mặc nhận các hoạt động được sắp xếp thứ tự theo thời gian kết thúc, hoạt động 1 có thời gian kết thúc sớm nhất. Ta muốn chứng tỏ có một giải pháp tối ưu bắt đầu bằng một chọn lựa tham, nghĩa là, với hoạt động 1.

Giả sử $A \subseteq S$ là một giải pháp tối ưu cho trường hợp đã cho của bài toán lựa chọn hoạt động, và ta hãy sắp xếp thứ tự các hoạt động trong A theo thời gian kết thúc. Giả sử thêm rằng hoạt động đầu tiên trong A là hoạt động k. Nếu $k = 1$, thì lịch trình A bắt đầu bằng một chọn lựa tham. Nếu $k \neq 1$, ta muốn chứng tỏ có một giải pháp tối ưu B khác cho S bắt đầu bằng chọn lựa tham, hoạt động 1. Cho $B = A - \{k\} \cup \{1\}$. Bởi $f_1 \leq f_k$, các hoạt động trong B rời nhau, và bởi B có cùng số lượng hoạt động như A, nên nó cũng tối ưu. Như vậy, B là một giải pháp tối ưu cho S chứa chọn lựa tham của hoạt động 1. Do đó, ta đã chứng tỏ luôn tồn tại một lịch trình tối ưu bắt đầu bằng một chọn lựa tham.

Hơn nữa, một khi thực hiện chọn lựa tham của hoạt động 1, bài toán sẽ thu gọn vào việc tìm một giải pháp tối ưu cho bài toán lựa chọn hoạt động trên những hoạt động trong S tương thích với hoạt động 1. Nghĩa là, nếu A là một giải pháp tối ưu cho bài toán ban đầu S, thì $A' = A - \{1\}$ là một giải pháp tối ưu cho bài toán lựa chọn hoạt động $S' = \{i \in S : s_i \geq f_1\}$. Tại sao? Nếu ta có thể tìm một giải pháp B' cho S' có các hoạt động nhiều hơn A', việc bổ sung hoạt động 1 vào B' sẽ mang lại một giải pháp B cho S có nhiều hoạt động hơn A, như vậy mâu thuẫn với tính tối ưu của A. Do đó, sau khi thực hiện từng chọn lựa tham, ta có một bài toán tối ưu hóa có dạng tương tự như bài toán ban đầu. Bằng phương pháp quy nạp trên số lượng các chọn lựa đã thực hiện, việc tiến hành

chọn lựa tham tại mọi bước sẽ tạo ra một giải pháp tối ưu.

Bài tập

17.1-1

Nêu một thuật toán lập trình động cho bài toán lựa chọn hoạt động, dựa trên tiến trình tính toán m_i lặp lại với $i = 1, 2, \ldots, n$, ở đó m_i là kích cỡ của tập hợp lớn nhất của các hoạt động tương thích lẫn nhau giữa các hoạt động $\{1, 2, \ldots, i\}$. Giả sử các nhập liệu đã được sắp xếp như trong phương trình (17.1). So sánh thời gian thực hiện của giải pháp với thời gian thực hiện của GREEDY-ACTIVITY-SELECTOR.

17.1-2

Giả sử ta có một tập hợp các hoạt động để lên lịch giữa một số lượng lớn các giảng đường. Ta muốn lên lịch tất cả các hoạt động sử dụng các giảng đường càng ít càng tốt. Nêu một thuật toán tham hiệu quả để xác định hoạt động nào sẽ dùng giảng đường nào.

(Điều này cũng được gọi là ***bài toán tô màu đồ thị quãng*** [interval-graph coloring problem]. Ta có thể tạo một đồ thị quãng có các đỉnh là các hoạt động đã cho và các cạnh của nó nối với các hoạt động không tương thích. Số lượng màu nhỏ nhất cần có để tô màu mọi đỉnh sao cho không có hai đỉnh kề nhau được gán cùng màu sẽ tương ứng với việc tìm số lượng giảng đường ít nhất cần có để lên lịch tất cả các hoạt động đã cho.)

17.1-3

Không phải bất kỳ cách tiếp cận tham nào cho bài toán lựa chọn hoạt động cũng đều tạo ra một tập hợp các hoạt động tương thích có kích cỡ cực đại. Hãy nêu một ví dụ để chứng tỏ không thể dùng cách tiếp cận lựa chọn hoạt động có thời hạn nhỏ nhất trong số các hoạt động tương thích với các hoạt động đã lựa chọn trước đó. Thực hiện như vậy với cách tiếp cận luôn lựa chọn hoạt động phủ chồng các hoạt động còn lại ít nhất.

17.2 Các thành phần của chiến lược tham

Một thuật toán tham có được một giải pháp tối ưu cho một bài toán bằng cách thực hiện một dãy các chọn lựa. Với mỗi điểm quyết định trong thuật toán, sự chọn lựa dường như tốt nhất hiện thời sẽ được chọn. Chiến lược phỏng đoán này không luôn tạo ra một giải pháp tối ưu, nhưng như đã thấy trong bài toán lựa chọn hoạt động, đôi lúc là đúng.

Đoạn này mô tả vài tính chất chung của các phương pháp tham.

Làm sao để có thể biết một thuật toán tham có giải quyết được một bài toán tối ưu hóa cụ thể hay không? Nói chung không có cách nào, nhưng có hai thành tố được biểu lộ bởi hầu hết các bài toán thích hợp với một chiến lược tham: tính chất của sự chọn lựa tham và cấu trúc con tối ưu.

Tính chất của sự chọn lựa tham

Thành tố chính đầu tiên là ***tính chất của sự chọn lựa tham*** [greedy-choice property]: một giải pháp tối ưu toàn cục có thể đạt được bằng cách thực hiện một chọn lựa (tham) tối ưu cục bộ. Đây là nơi các thuật toán tham khác với lập trình động. Trong lập trình động, ta thực hiện một chọn lựa tại từng bước, nhưng sự chọn lựa có thể tùy thuộc vào các giải pháp cho các bài toán con. Trong một thuật toán tham, ta thực hiện bất kỳ chọn lựa nào có vẻ tốt nhất hiện thời rồi giải quyết các bài toán con nảy sinh sau khi thực hiện sự chọn lựa. Chọn lựa mà một thuật toán tham thực hiện có thể tùy thuộc vào các chọn lựa cho đến giờ, nhưng nó không thể tùy thuộc vào bất kỳ chọn lựanào trong tương lai hoặc vào các giải pháp cho các bài toán con. Như vậy, khác với lập trình động, thường giải quyết các bài toán con từ dưới lên, một chiến lược tham thường tiến triển theo cách trên xuống, thực hiện một chọn lựa tham sau một chọn lựa khác, liên tục rút gọn từng trường hợp bài toán đã cho thành một bài toán nhỏ hơn.

Tất nhiên, ta phải chứng minh rằng một chọn lựa tham tại từng bước sẽ cho ra một giải pháp tối ưu toàn cục, và đây là nơi cần có sự thông minh. Thông thường, như trong trường hợp của Định lý 17.1, phần chứng minh xem xét một giải pháp tối ưu toàn cục. Như vậy nó chứng tỏ có thể sửa đổi giải pháp sao cho một chọn lựa tham được thực hiện làm bước đầu tiên, và chọn lựa này rút gọn bài toán thành một bài toán tương tự song nhỏ hơn. Như vậy, phương pháp quy nạp được áp dụng để chứng tỏ có thể dùng một chọn lựa tham tại mọi bước. Việc chứng tỏ một chọn lựa tham dẫn đến một bài toán tương tự song nhỏ hơn thường rút gọn phần chứng minh tính đúng đắn thành tiến trình chứng minh một giải pháp tối ưu phải biểu lộ cấu trúc con tối ưu.

Cấu trúc con tối ưu

Một bài toán biểu lộ *cấu trúc con tối ưu* nếu một giải pháp tối ưu của bài toán chứa trong nó các giải pháp tối ưu cho các bài toán con. Tính chất này là thành tố chính để ước định khả năng áp dụng của lập trình động cũng như các thuật toán tham. Để lấy ví dụ về cấu trúc con tối ưu, ta nhớ lại phần chứng minh của Định lý 17.1 đã chứng minh rằng nếu

một giải pháp tối ưu A cho bài toán lựa chọn hoạt động bắt đầu bằng hoạt động 1, thì tập hợp các hoạt động $A' = A - \{1\}$ là một giải pháp tối ưu cho bài toán lựa chọn hoạt động $S' = \{i \in S: s_i \geq f_1\}$.

Tham đối lại lập trình động

Do tính chất cấu trúc con tối ưu được cả chiến lược lập trình động lẫn chiến lược tham khai thác, nên có thể ta có khuynh hướng phát sinh một giải pháp lập trình động cho một bài toán khi chỉ một giải pháp tham là đủ, hoặc ta có thể lầm lẫn cho rằng một giải pháp tham sẽ làm việc khi thực tế lại cần một giải pháp lập trình động. Để minh họa những điểm tinh tế giữa hai kỹ thuật, ta hãy nghiên cứu hai viến thể của một bài toán tối ưu hóa cổ điển.

Bài toán ba lô 0-1 được đặt ra như sau. Một tên trộm đánh cắp một cửa hiệu tìm thấy n mục; mục thứ i đánh giá v_i đôla và cân nặng w_i pound, ở đó v_i và w_i là các số nguyên. Hắn ta muốn lấy một tải trọng càng có giá càng tốt, nhưng hắn có thể mang tối đa W pound trong ba lô của mình cho vài số nguyên W. Hắn sẽ lấy những mục nào? (Điều này được gọi là bài toán ba lô 0-1 bởi mỗi mục phải được lấy hoặc để lại đàng sau; tên trộm không thể lấy một lượng phân số của một mục hoặc lấy một mục nhiều lần.)

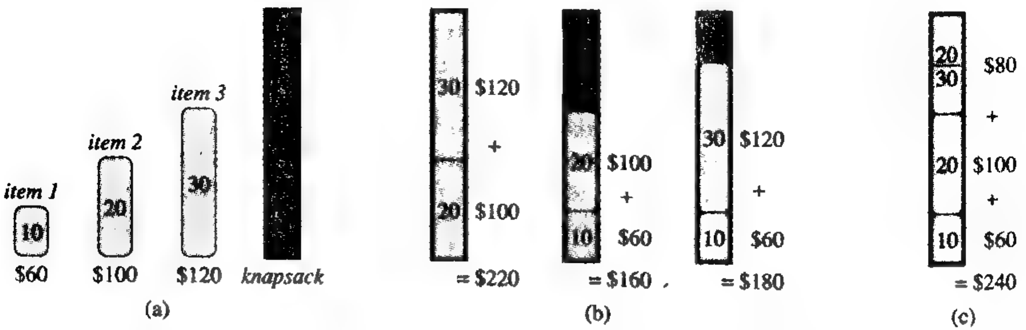
Trong ***bài toán ba lô phân số***, tiến trình xác lập cũng vậy, nhưng tên trộm có thể lấy các phần phân số của các mục, thay vì phải thực hiện một chọn lựa nhị phân (0-1) cho mỗi mục. Bạn có thể nghĩ một mục trong bài toán ba lô 0-1 giống như một thỏi vàng, trong khi một mục trong bài toán ba lô phân số giống như bụi vàng.

Cả hai bài toán ba lô biểu lộ tính chất cấu trúc con tối ưu. Với bài toán 0-1, ta hãy xét tải trọng có giá trị nhất cân nặng tối đa W pound. Nếu gỡ bỏ mục j ra khỏi tải trọng này, tải trọng còn lại phải là tải trọng có giá trị nhất cân nặng tối đa $W - w_j$ mà tên trộm có thể lấy từ $n - 1$ mục ban đầu loại trừ j. Với bài toán phân số có thể so sánh, xét rằng nếu ta gỡ bỏ một trọng lượng w của một mục j ra khỏi tải trọng tối ưu, tải trọng còn lại phải là tải trọng có giá trị nhất cân nặng tối đa $W - w$ mà tên trộm có thể lấy từ $n - 1$ mục ban đầu cộng với $w_j - w$ pound của mục j.

Tuy các bài toán là tương tự, song bài toán ba lô phân số có thể giải quyết bằng chiến lược tham, trong khi đó bài toán 0-1 lại không thể. Để giải quyết bài toán phân số, trước tiên ta tính toán giá trị mỗi pound v_i/w_i cho từng mục. Tuân thủ chiến lược tham, tên trộm bắt đầu bằng cách lấy càng nhiều càng tốt mục có giá trị mỗi pound lớn nhất. Nếu nguồn cung cấp mục đó cạn kiệt và hắn ta vẫn có thể mang thêm, hắn lấy càng nhiều càng tốt mục có giá trị mỗi pound lớn nhất kế tiếp, và vân vân

cho đến khi không thể mang thêm gì được nữa. Như vậy, bằng cách sắp xếp các mục theo giá trị mỗi pound, thuật toán tham chạy trong $O(n \lg n)$ thời gian. Phần chứng minh bài toán ba lô phân số có tính chất của sự chọn lựa tham được để lại cho Bài tập 17.2-1.

Để chứng tỏ chiến lược tham này không làm việc cho bài toán ba lô 0-1, ta xét trường hợp bài toán minh họa trong Hình 17.2(a). Có 3 mục, và ba lô có thể lưu giữ 50 pound. Mục 1 cân nặng 10 pound và đáng giá 60 đôla. Mục 2 cân nặng 20 pound và đáng giá 100 đôla. Mục 3 cân nặng 30 pound và đáng giá 120 đôla. Như vậy, giá trị mỗi pound của mục 1 là 6 đôla mỗi pound, lớn hơn giá trị mỗi pound của mục 2 (5 đôla mỗi pound) hoặc mục 3 (4 đôla mỗi pound). Do đó, chiến lược tham sẽ lấy mục 1 trước. Tuy nhiên, như sẽ thấy trong phần phân tích trường hợp ở Hình 17.2(b), giải pháp tối ưu lấy các mục 2 và 3, để lại 1 đàng sau. Hai giải pháp khả dĩ liên quan đến mục 1 đều là tối ưu con [suboptimal].



Hình 17.2 Chiến lược tham không làm việc cho bài toán ba lô 0-1. **(a)** Tên trộm phải lựa một tập hợp con ba mục được nêu có trọng lượng không được vượt quá 50 pound. **(b)** Tập hợp con tối ưu bao gồm các mục 2 và 3. Mọi giải pháp có mục 1 đều là tối ưu con, cho dù mục 1 có giá trị lớn nhất mỗi pound. **(c)** Với bài toán ba lô phân số, việc lấy các mục theo thứ tự của giá trị lớn nhất mỗi pound sẽ cho ra một giải pháp tối ưu.

Tuy nhiên, với bài toán phân số có thể so sánh, chiến lược tham, lấy mục 1 trước, sẽ mang lại một giải pháp tối ưu, như đã nêu trong Hình 17.2(c). Việc lấy mục 1 không làm việc trong bài toán 0-1 bởi tên trộm không thể nhét đầy ắp ba lô của mình, và không gian trống hạ thấp giá trị hiệu quả mỗi pound của tải trọng của hắn. Trong bài toán 0-1, khi ta xét một mục để gộp vào ba lô, ta phải so sánh giải pháp cho bài toán con ở đó mục được gộp với giải pháp cho bài toán con ở đó mục được

loại trừ trước khi ta có thể thực hiện sự chọn lựa. Bài toán được lập thành theo cách này gây ra nhiều bài toán con phủ chồng—một dấu chất lượng của lập trình động, và quả thực, lập trình động có thể được dùng để giải quyết bài toán 0-1. (Xem Bài tập 17.2-2.)

Bài tập

17.2-1

Chứng minh bài toán ba lô phân số có tính chất của sự chọn lựa tham.

17.2-2

Nêu một giải pháp lập trình động cho bài toán ba lô 0-1 chạy trong $O(n\ W)$ thời gian, ở đó n là số lượng các mục và W là trọng lượng cực đại của các mục mà tên trộm có thể mang trong ba lô của hắn.

17.2-3

Giả sử trong một bài toán ba lô 0-1, thứ tự các mục khi được sắp xếp theo trọng lượng tăng giống như thứ tự của chúng khi được sắp xếp theo giá trị giảm. Nêu một thuật toán hiệu quả để tìm một giải pháp tối ưu cho biến thể này của bài toán ba lô, và lập luận rằng thuật toán là đúng đắn.

17.2-4

Giáo sư Midas lái một chiếc xe từ Newark đến Reno dọc theo Interstate 80. Thùng xăng xe của ông, khi đầy, chứa đủ xăng để chạy n dặm, và bản đồ của ông cho thấy các khoảng cách giữa các trạm xăng trên tuyến đường của ông. Dọc đường đi, giáo sư muốn dừng càng ít điểm đổ xăng càng tốt. Nêu một phương pháp hiệu quả qua đó giáo sư Midas có thể xác định ông sẽ dừng tại những trạm xăng nào, và chứng minh rằng chiến lược của bạn sẽ cho ra một giải pháp tối ưu.

17.2-5

Mô tả một thuật toán hiệu quả mà, căn cứ vào một tập hợp $\{x_1, x_2, \ldots, x_n\}$ các điểm trên đường thẳng thực, xác định tập hợp nhỏ nhất gồm các quãng đóng có chiều dài đơn vị chứa tất cả các điểm đã cho. Chứng tỏ thuật toán của bạn là đúng.

17.2-6 *

Nêu cách giải quyết bài toán ba lô phân số trong $O(n)$ thời gian. Giả sử bạn có một giải pháp cho Bài toán 10-2.

17.3 Các mã Huffman

Các mã Huffman là một kỹ thuật được dùng rộng rãi và rất hiệu quả để nén dữ liệu; thường tiết kiệm được từ 20% đến 90%, tùy theo các đặc tính của tập tin đang được nén. Thuật toán tham của Huffman sử dụng một bảng các tần số lần xuất hiện của từng ký tự để xây dựng nên một phương cách tối ưu nhằm biểu thị từng ký tự dưới dạng chuỗinhị phân.

Giả sử có một tập tin dữ liệu 100,000 ký tự mà ta muốn lưu trữ ở dạng nén. Ta nhận thấy các ký tự trong tập tin xảy ra với các tần số mà Hình 17.3 đã cho. Nghĩa là, chỉ xuất hiện sáu ký tự khác nhau, và ký tự a xảy ra 45,000 lần.

Có nhiều cách để biểu diễn một tập tin thông tin như vậy. Ta xét bài toán thiết kế ***mã ký tự nhị phân*** (hoặc gọi tắt là ***mã***) ở đó mỗi ký tự được biểu thị bởi một chuỗi nhị phân duy nhất. Nếu dùng một ***mã có chiều dài cố định***, ta cần 3 bit để biểu thị sáu ký tự: a = 000, b = 001, ..., f = 101. Phương pháp này yêu cầu 300,000 bit để mã hóa nguyên cả tập tin. Có thể làm tốt hơn không?

	a	b	c	d	e	f
Tần số (theo cấp ngàn)	45	13	12	16	9	5
Từ mã có chiều dài cố định	000	001	010	011	100	101
Từ mã có chiều dài biến đổi	0	101	100	111	1101	1100

Hình 17.3 Bài toán mã hóa ký tự. Một tập tin dữ liệu gồm 100,000 ký tự chỉ chứa các ký tự a-f, có các tần số đã chỉ rõ. Nếu mỗi ký tự được gán một từ mã 3 bit, tập tin có thể được mã hóa trong 300,000 bit. Dùng mã có chiều dài biến đổi đã nêu, tập tin có thể được mã hóa trong 224,000 bit.

Một ***mã có chiều dài biến đổi*** có thể thực hiện tốt hơn đáng kể so với mã có chiều dài cố định, nhờ gán cho các ký tự thường xuyên các từ mã ngắn và các ký tự không thường xuyên các từ mã dài. Hình 17.3 có nêu một kiểu mã như vậy; ở đây chuỗi 1-bit 0 biểu thị cho a, và chuỗi 4-bit 1100 biểu thị chi f. Mã này yêu cầu

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1{,}000 = 224{,}000 \text{ bit}$$

để biểu thị tập tin, một khoản tiết kiệm xấp xỉ 25%. Thực vậy, đây là một mã ký tự tối ưu cho tập tin này, như sẽ thấy.

Các mã tiền tố

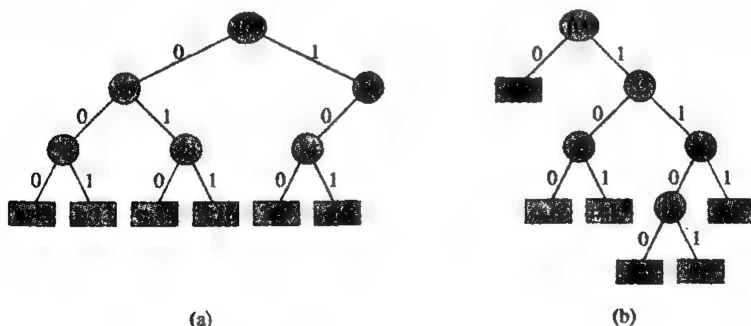
Trong trường hợp này ta chỉ xét các mã ở đó không có từ mã nào là một tiền tố của vài từ mã khác. Các mã như vậy được gọi là ***mã tiền tố***

[prefix codes][1]. Tuy không thực hiện ở đây, song ta có thể chứng tỏ tính năng nén dữ liệu tối ưu mà một mã ký tự có thể đạt được sẽ luôn có thể hoàn thành bằng một mã tiền tố, do đó không bị mất đi tính tổng quát trong việc hạn chế sự chú ý vào các mã tiền tố.

Các mã tiền tố là thỏa đáng bởi chúng rút gọn tiến trình mã hóa (nén) và giải mã. Tiến trình mã hóa luôn đơn giản với bất kỳ mã ký tự nhị phân nào; ta chỉ việc ghép nối các từ mã biểu thị cho mỗi ký tự của tập tin. Ví dụ, với mã tiền tố có chiều dài biến đổi của Hình 17.3, ta mã hóa tập tin 3-ký tự abc dưới dạng 0 • 101 • 100 = 0101100, ở đó ta dùng "•" để thể hiện phép ghép nối.

Tiến trình giải mã cũng khá đơn giản với một mã tiền tố. Do không có từ mã nào là một tiền tố của một từ mã khác, nên từ mã bắt đầu một tập tin đã mã hóa thường không mơ hồ. Ta có thể đơn giản định danh từ mã ban đầu, phiên dịch nó trở về ký tự ban đầu, gỡ bỏ nó ra khỏi tập tin đã mã hóa, và lặp lại tiến trình giải mã trên phần còn lại của tập tin đã mã hóa. Trong ví dụ của chúng ta, chuỗi 001011101 phân ngữ duy nhất dưới dạng 0 • 0 • 101 • 1101, sẽ giải mã thành aabe.

Tiến trình giải mã cần một phần biểu diễn tiện dụng cho mã tiền tố sao cho có thể dễ dàng chọn ra từ mã ban đầu. Một cây nhị phân có các lá là các ký tự đã cho sẽ cung cấp một phần biểu diễn như vậy. Ta diễn dịch từ mã nhị phân của một ký tự dưới dạng lộ trình từ gốc đến ký tự đó, ở đó 0 có nghĩa là "đi đến con trái" và 1 có nghĩa là "đi đến con phải." Hình 17.4 nêu các cây cho hai mã của ví dụ chúng ta. Lưu ý, chúng không phải là các cây tìm nhị phân, bởi các lá không cần xuất hiện theo thứ tự sắp xếp và các nút trong không chứa các khóa ký tự.



Hình 17.4 Các cây tương ứng với các lược đồ mã hóa trong Hình 17.3. Mỗi lá được gán nhãn bằng một ký tự và tần số lần xuất hiện của nó. Mỗi nút trong được gán nhãn bằng tổng các trọng số của các lá trong cây con của nó. **(a)** Cây tương ứng với mã có chiều dài cố định a = 000,..., f = 100. **(b)** Cây tương ứng với mã tiền tố tối ưu a = 0, b = 101,..., f = 1100.

[1] *Có lẽ tốt hơn ta nên gọi là "các mã không có tiền tố", nhưng thuật ngữ "mã tiền tố" là chuẩn trong các tài liệu giáo khoa.*

Một mã tối ưu cho một tập tin luôn được biểu diễn bởi một cay nhị phân *đầy đủ*, ở đó mọi nút không lá đều có hai con (xem Bài tập 17.3-1). Mã có chiều dài cố định trong ví dụ của chúng ta không tối ưu bởi cây của nó, xem Hình 17.4(a), không phải là một cây nhị phân đầy đủ: có các từ mã bắt đầu 10..., nhưng không có từ mã nào bắt đầu 11.... Do giờ đây có thể hạn chế sự chú ý vào các cây nhị phân đầy đủ, nên ta có thể nói rằng nếu C là bảng chữ cái từ đó các ký tự được rút ra, thì cây cho một mã tiền tố tối ưu có chính xác $|C|$ lá, Mỗi lá cho một mẫu tự của bảng chữ cái, và chính xác $|C| - 1$ nút trong.

Cho một cây T tương ứng với một mã tiền tố, ta có thể đơn giản tính toán số lượng bit cần thiết để mã hóa một tập tin. Với mỗi ký tự c trong bảng chữ cái C, cho $f(c)$ thể hiện tần số của c trong tập tin và cho $d_T(c)$ thể hiện chiều sâu lá của c trong cây. Lưu ý, $d_T(c)$ cũng là chiều dài từ mã cho ký tự c. Như vậy, số lượng bit cần thiết để mã hóa một tập tin là

$$B(T) = \sum_{c \in C} f(c) d_T(c) , \qquad (17.3)$$

mà ta định nghĩa là ***mức hao phí*** của cây T.

Kiến tạo một mã Huffman

Huffman đã sáng chế một thuật toán tham liên tục một mã tiền tố tối ưu có tên ***mã Huffman.*** Thuật toán xây dựng cây T tương ứng với mã tối ưu theo kiểu dưới lên. Nó bắt đầu bằng một tập hợp $|C|$ lá và thực hiện một dãy $|C| - 1$ các phép toán "trộn" để tạo cây chung cuộc.

Trong mã giả dưới đây, ta mặc nhận C là một tập hợp n ký tự và mỗi ký tự $c \in C$ là một đối tượng có một tần số đã định nghĩa $f[c]$.

Một hàng đợi ưu tiên Q, được lập khóa trên f, được dùng để định danh hai đối tượng ít thường xuyên nhất để trộn với nhau. Kết quả của phép trộn hai đối tượng là một đối tượng mới có tần số là tổng của các tần số của hai đối tượng được trộn.

HUFFMAN(C)

1 $n \leftarrow |C|$

2 $Q \leftarrow C$

3 **for** $i \leftarrow 1$ **to** $n - 1$

```
4      do z ← ALLOCATE-NODE()
5           x ← left[z] ← EXTRACT-MIN(Q)
6           y ← right[z] ← EXTRACT-MIN(Q)
7           f[z] ← f[x] + f[y]
8           INSERT(Q, z)
9   return EXTRACT-MIN(Q)
```

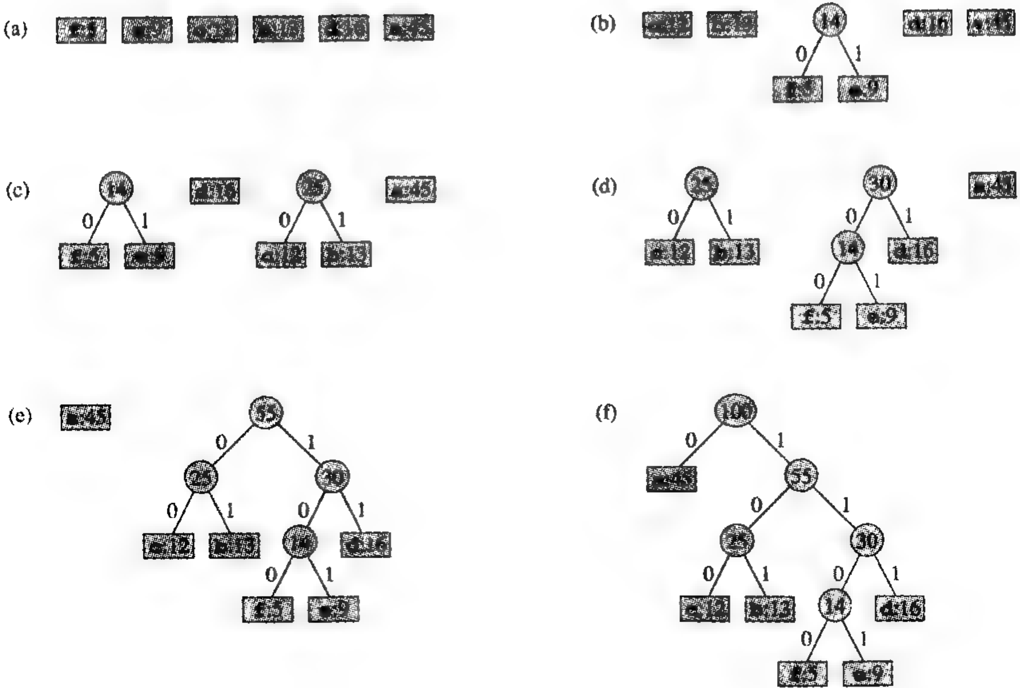
Với ví dụ của chúng ta, thuật toán của Huffman tiến hành như đã nêu trong Hình 17.5. Do có 6 mẫu tự trong bảng chữ cái, kích cỡ hàng đợi ban đầu là $n = 6$, và cần có 5 bước trộn để xây dựng cây. Cây chung cuộc biểu diễn mã tiền tố tối ưu. Từ mã cho một mẫu tự là dãy các nhãn cạnh trên lộ trình từ gốc đến mẫu tự đó.

Dòng 2 khởi tạo hàng đợi ưu tiên Q có các ký tự trong C. Vòng lặp **for** trong các dòng 3-8 liên tục trích hai nút x và y có tần số thấp nhất từ hàng đợi, và thay chúng trong hàng đợi bằng một nút mới z biểu thị cho phép trộn của chúng. Tần số của z được tính toán dưới dạng tổng của các tần số của x và y trong dòng 7. Nút z có x làm con trái và y làm con phải của nó. (Thứ tự này là tùy ý; việc chuyển con trái và con phải của một nút bất kỳ sẽ cho ra một mã khác có cùng mức hao phí.) Sau n - 1 phép trộn, nút một được để lại trong hàng đợi—gốc của cây mã—được trả về trong dòng 9.

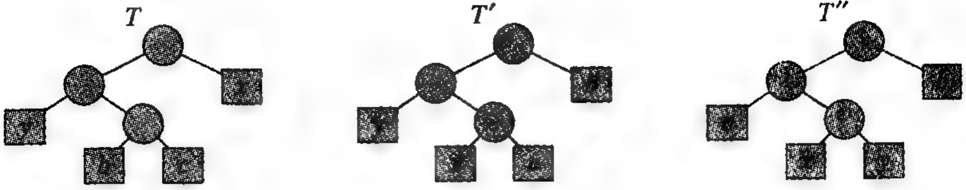
Đợt phân tích thời gian thực hiện của thuật toán Huffman mặc nhận Q được thực thi dưới dạng một đống nhị phân (xem Chương 7). Với một tập hợp C n ký tự, việc khởi tạo của Q trong dòng 2 có thể được thực hiện trong $O(n)$ thời gian dùng thủ tục BUILD-HEAP trong Đoạn 7.3. Vòng lặp **for** trong các dòng 3-8 được thi hành chính xác $|n|$ - 1 lần, và do mỗi phép toán đống yêu cầu thời gian $O(\lg n)$, nên vòng lặp đóng góp $O(n \lg n)$ vào thời gian thực hiện. Như vậy, tổng thời gian thực hiện của HUFFMAN trên một tập hợp n ký tự là $O(n \lg n)$.

Tính đúng đắn của thuật toán Huffman

Để chứng minh thuật toán tham HUFFMAN là đúng đắn, ta chứng tỏ bài toán xác định một mã tiền tố tối ưu biểu lộ sự chọn lựa tham và các tính chất cấu trúc con tối ưu. Bổ đề dưới đây chứng tỏ tính chất của sự chọn lựa tham vẫn có hiệu lực.



Hình 17.5 Các bước của thuật toán Huffman cho các tần số đã cho trong Hình 17.3. Mỗi phần nêu nội dung của hàng đợi được sắp xếp theo thứ tự tăng dần theo tần số. Tại mỗi bước, hai cây có các tần số thấp nhất được trộn. Các lá được nêu dưới dạng các hình chữ nhật chứa một ký tự và tần số của nó. Các nút trong được nêu dưới dạng các vòng tròn chứa tổng các tần số của các con của nó. Mỗi cạnh nối một nút trong với các con của nó sẽ được gán nhãn 0 nếu nó là một cạnh đến một con trái và 1 nếu nó là một cạnh đến một con phải. Từ mã của một mẫu tự là dãy các nhãn trên các cạnh nối gốc với lá của mẫu tự đó. **(a)** tập hợp ban đầu $n = 6$ nút, mỗi nút cho một mẫu tự. **(b)**-**(e)** Các giai đoạn trung gian. **(f)** Cây chung cuộc.



Hình 17.6 Minh họa bước chính trong phần chứng minh của Bổ đề 17.2. Trong cây T tối ưu, các lá b và c là hai trong số các lá sâu nhất và là anh em ruột. Các lá x và y là hai lá mà thuật toán Huffman trộn với nhau đầu tiên; chúng xuất hiện theo các vị trí tùy ý trong T. Các lá b và x được tráo để được cây T'. Sau đó, các lá c và y được tráo để được cây T''. Bởi mỗi lần tráo không làm tăng mức hao phí, nên cây T'' kết quả cũng là một cây tối ưu.

Bổ đề 17.2

Cho C là một bảng chữ cái ở đó mỗi ký tự $c \in C$ có tần số $f[c]$. Cho x và y là hai ký tự trong C có các tần số thấp nhất. Như vậy sẽ tồn tại một mã tiền tố tối ưu cho C ở đó các từ mã của x và y có cùng chiều dài và chỉ khác trong bit cuối.

Chứng minh Ý tưởng của phần chứng minh đó là lấy cây T biểu thị cho một mã tiền tố tối ưu tùy ý và sửa đổi nó để kiến một cây biểu thị cho một mã tiền tố tối ưu khác sao cho các ký tự x và y xuất hiện dưới dạng các lá anh em ruột có chiều sâu cực đại trong cây mới. Nếu ta có thể thực hiện điều này, thì các từ mã của chúng sẽ có cùng chiều dài và chỉ khác trong bit cuối.

Cho b và c là hai ký tự là các lá anh em ruột có chiều sâu cực đại trong T. Không để mất tính tổng quát, ta mặc nhận $f[b] \le f[c]$ và $f[x] \le f[y]$. Bởi $f[x]$ và $f[y]$ là hai tần số lá thấp nhất, theo thứ tự, và $f[b]$ và $f[c]$ là hai tần số tùy ý, theo thứ tự, nên ta có $f[x] \le f[b]$ và $f[y] \le f[c]$. Như đã nêu trong Hình 17.6, ta trao đổi các vị trí trong T của b và x để tạo ra một cây T', rồi trao đổi các vị trí trong T' của c và y để tạo ra cây T''. Qua phương trình (17.3), sự khác biệt về mức hao phí giữa T và T' là

$$\begin{aligned}
B(T) - B(T') &= \sum_{c\in C} f(c)d_T(c) - \sum_{c\in C} f(c)d_{T'}(c) \\
&= f[x]d_T(x) + f[b]d_T(b) - f[x]d_{T'}(x) - f[b]d_{T'}(b) \\
&= f[x]d_T(x) + f[b]d_T(b) - f[x]d_T(b) - f[b]d_T(x) \\
&= (f[b] - f[x](d_T(b) - d_T(x)) \\
&\ge 0,
\end{aligned}$$

bởi cả $f[b] - f[x]$ và $d_T[b] - d_T[x]$ đều không âm. Cụ thể hơn, $f[b] - f[x]$ là không âm bởi x là một lá có tần số cực tiểu, và $d_T[b] - d_T[x]$ không âm bởi b là một lá có chiều sâu cực đại trong T. Cũng vậy, do sự trao đổi y và c không làm tăng mức hao phí, nên $B(T') - B(T'')$ là không âm. Do đó, $B(T'') \le B(T)$, và bởi T là tối ưu, nên $B(T) \le B(T'')$, hàm ý $B(T'') = B(T)$. Như vậy, T'' là một cây tối ưu ở đó x và y xuất hiện dưới dạng các lá anh em ruột có chiều sâu cực đại, mà bổ đề theo.

Bổ đề 17.2 hàm ý tiến trình tạo dựng một cây tối ưu bằng các phép hợp nhất có thể, mà không làm mất tính tổng quát, bắt đầu bằng một chọn lựa tham của tiến trình trộn hai ký tự có tần số thấp nhất đó với nhau. Tại sao đây là một chọn lựa tham? Ta có thể xem mức hao phí của một phép hợp nhất đơn lẻ dưới dạng tổng các tần số của hai mục đang được hợp nhất. Bài tập 17.3-3 chứng tỏ tổng mức hao phí của cây được kiến tạo chính là tổng các mức hao phí của các phép hợp nhất của nó. Từ tất cả các phép hợp nhất khả dĩ tại mỗi bước, HUFFMAN chọn phép toán gánh chịu mức hao phí ít nhất.

Bổ đề sau đây chứng tỏ bài toán kiến tạo các mã tiền tố tối ưu có tính chất cấu trúc con tối ưu.

Bổ đề 17.3

Cho T là một cây nhị phân đầy đủ biểu thị cho mã tiền tố tối ưu trên một bảng chữ cái C, ở đó tần số $f[c]$ được định nghĩa cho mỗi ký tự $c \in C$. Xét hai ký tự x và y bất kỳ xuất hiện dưới dạng các lá anh em ruột trong T, và cho z là cha của chúng. Sau đó, xét z dưới dạng một ký tự có tần số $f[z] = f[x] + f[y]$, cây $T' = T - \{x,y\}$ biểu diễn cho một mã tiền tố tối ưu cho bảng chữ cái $C' = C - \{x, y\} \cup \{z\}$.

Chứng minh Trước tiên ta chứng tỏ mức hao phí $B(T)$ của cây T có thể được diễn tả theo dạng mức hao phí $B(T')$ của cây T' bằng cách xét các mức hao phí thành phần trong phương trình (17.3). Với mỗi $c \in C - \{x, y\}$, ta có $d_T(c) = d_{T'}(c)$, và do đó $f[c]d_T(c) = f[c]d_{T'}(c)$. Bởi $d_T(x) = d_T(y) = d_{T'}(z) + 1$, ta có

$$\begin{aligned} f[x]d_T(x) + f[y]d_T(y) &= (f[x] + f[y])(d_{T'}(z) + 1) \\ &= f[z]d_{T'}(z) + (f[x] + f[y]), \end{aligned}$$

từ đó ta kết luận rằng

$$B(T) = B(T') + f[x] + f[y].$$

Nếu T' biểu diễn cho một mã tiền tố không tối ưu cho bảng chữ cái C', thì ở đó tồn tại một cây T'' có các lá là các ký tự trong C' sao cho $B(T'') < B(T')$. Bởi z được xem như một ký tự trong C', nên nó xuất hiện dưới dạng một lá trong T''. Nếu ta cộng x và y dưới dạng các con của z

trong T', ta được một mã tiền tố cho C với mức hao phí $B(T'') + f[x] + f[y] < B(T)$, trái ngược với tính tối ưu của T. Như vậy, T' phải là tối ưu cho bảng chữ cái C'.

Định lý 17.4

Thủ tục HUFFMAN tạo ra một mã tiền tố tối ưu.

Chứng minh Tức thời từ các Bổ đề 17.2 và 17.3.

Bài tập

17.3-1

Chứng minh một cây nhị phân không đầy không thể tương ứng với một mã tiền tố tối ưu.

17.3-2

Đâu là một mã Huffman tối ưu cho tập hợp các tần số dưới đây, dựa trên 8 số Fibonacci đầu tiên?

a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21

Có thể tổng quát hóa đáp án để tìm ra mã tối ưu khi các tần số là n số Fibonacci đầu tiên không?

17.3-3

Chứng minh tổng mức hao phí của một cây cho một mã cũng có thể được tính toán dưới dạng tổng, trên tất cả các nút trong, của các tần số kết hợp của hai con của nút đó.

17.3-4

Chứng minh rằng với một mã tối ưu, nếu các ký tự được sắp xếp thứ tự sao cho các tần số của chúng không tăng, thì các chiều dài từ mã của chúng không giảm.

17.3-5

Cho $C = \{0, 1, ..., n - 1\}$ là một tập hợp các ký tự. Chứng tỏ bất kỳ mã tiền tố tối ưu nào trên C đều có thể được biểu diễn bằng một dãy

$2n - 1 + n\lceil \lg n \rceil$

bit. (*Mách nước:* Dùng $2n$-1 bit để đặc tả cấu trúc của cây, như được phát hiện bởi một tầng của cây.)

17.3-6

Hãy tổng quát hóa thuật toán Huffman để tam phân các từ mã (tức là,

các từ mã dùng các the các ký hiệu 0, 1, và 2), và chứng minh nó cho ra các mã tam phân tối ưu.

17.3-7

Giả sử một tập tin dữ liệu chứa một dãy các ký tự 8-bit sao cho tất cả ký tự 256 có mặt dưới dạng chung: tần số ký tự cực đại nhỏ hơn hai lần tần số ký tự cực tiểu. Chứng minh phương pháp lập mã Huffman trong trường hợp này sẽ không hiệu quả hơn so với việc dùng một mã cố định có chiều dài 8-bit bình thường.

17.3-8

Chứng tỏ không thể dự kiến một lược đồ nén để nén một tập tin gồm các ký tự 8-bit được chọn ngẫu nhiên theo thậm chí một bit đơn lẻ. (*Mách nước:* So sánh số lượng tập tin với số lượng các tập tin mã hóa khả dĩ.)

---

* 17.4 Nền tảng lý thuyết cho các phương pháp tham

Có một lý thuyết hay về các thuật toán tham, mà ta phác họa trong đoạn này. Lý thuyết này tỏ ra hữu ích trong việc xác định khi nào phương pháp tham cho ra các giải pháp tối ưu. Nó liên quan đến các cấu trúc tổ hợp mệnh danh là "tạng ma trận" [matroids]. Tuy lý thuyết này không đề cập tất cả các trường hợp mà một phương pháp tham áp dụng (ví dụ, nó không đề cập bài toán lựa chọn hoạt động của Đoạn 17.1 hoặc bài toán mã hóa Huffman của Đoạn 17.3), song nó đề cập nhiều trường hợp có lợi ích thực tiễn. Vả lại, lý thuyết này đang được nhanh chóng phát triển và mở rộng để bao hàm nhiều ứng dụng hơn; xem các ghi chú cuối chương này để có các tham khảo.

17.4.1 Các tạng ma trận

Một ***tạng ma trận*** [matroid] là một cặp có thứ tự $M = (S, \mathcal{I})$ thỏa các điều kiện dưới đây.

1. S là một tập hợp không trống hữu hạn.

2. $\mathcal{I}$ là một gia đình không trống gồm các tập hợp con của S, có tên các tập hợp con ***độc lập*** của S, sao cho nếu $B \in \mathcal{I}$ và $A \subseteq B$, thì $A \in \mathcal{I}$. Ta nói rằng $\mathcal{I}$ là ***di truyền*** nếu nó thỏa tính chất này. Lưu ý, tập hợp trống $\emptyset$ nhất thiết phải là một phần tử của $\mathcal{I}$.

3. Nếu $A \in \mathcal{I}$, $B \in \mathcal{I}$, và $|A| < |B|$, thì có một thành phần $x \in B - A$ sao cho $A \cup \{x\} \in \mathcal{I}$. Ta nói rằng M thỏa ***tính chất trao đổi***.

Từ "matroid" do Hassler Whitney đặt. Ông đã nghiên cứu các ***tạng ma trận matric*** [matric matroids], ở đó các thành phần của S là các hàng của một ma trận đã cho và một tập hợp các hàng là độc lập nếu chúng độc lập một cách tuyến tính theo nghĩa thông thường. Có thể dễ dàng chứng tỏ rằng cấu trúc này định nghĩa một tạng ma trận (xem Bài tập 17.4-2).

Để minh họa thêm các tạng ma trận, ta xét ***tạng ma trận đồ họa*** $M_G = (S_G, \mathcal{I}_G)$ được định nghĩa theo dạng một đồ thị không có hướng đã cho $G = (V, E)$ như sau.

- Tập hợp S_G được định nghĩa là E, tập hợp các cạnh của G.
- Nếu A là một tập hợp con của E, thì $A \in \mathcal{I}_G$ nếu và chỉ nếu A là phi chu trình. Nghĩa là, một tập hợp các cạnh là độc lập nếu và chỉ nếu nó hình thành một rừng.

Tạng ma trận đồ họa M_G liên quan mật thiết đến bài toán cây tỏa nhánh cực tiểu, đã đề cập chi tiết trong Chương 24.

***Định lý* 17.5**

Nếu G là một đồ thị không có hướng, thì $M_G = (S_G, \mathcal{I}_G)$ là một tạng ma trận.

Chứng minh Rõ ràng, $S_G = E$ là một tập hợp hữu hạn. Vả lại, $\mathcal{I}_G$ là bắc cầu, bởi một tập hợp con của một rừng là một rừng. Nói một cách khác, việc gỡ bỏ các cạnh ra khỏi một tập hợp phi chu trình các cạnh không thể tạo các chu trình.

Như vậy, ta chỉ cần chứng tỏ M_G thỏa tính chất trao đổi. Giả sử A và B là các rừng của G và rằng $|B| > |A|$. Nghĩa là, A và B là các tập hợp phi chu trình các cạnh, và B chứa nhiều cạnh hơn A.

Theo Định lý 5.2, một rừng có k cạnh chứa chính xác $|V|-k$ cây. (Để chứng minh điều này một cách khác, ta hãy bắt đầu bằng $|V|$ cây và không có cạnh nào. Sau đó, mỗi cạnh được bổ sung vào rừng sẽ làm giảm số lượng cây là một.) Như vậy, rừng A chứa $|V| - |A|$ cây, và rừng B chứa $|V| - |B|$ cây.

Do rừng B có ít cây hơn rừng A, nên rừng B phải chứa một cây T có các đỉnh nằm trong hai cây khác nhau trong rừng A. Hơn nữa, bởi T được liên thông, nên nó phải chứa một cạnh (u, v) sao cho các đỉnh u và v nằm trong các cây khác nhau trong rừng A. Bởi cạnh (u, v) liên thông các đỉnh trong hai cây khác nhau trong rừng A, nên cạnh (u, v) có thể được bổ sung rừng A mà không tạo một chu trình. Do đó, M_G thỏa tính chất trao đổi, hoàn tất phần chứng minh M_G là một tạng ma trận.

Cho một tạng ma trận $M = (S, \mathcal{I})$, ta gọi một thành phần $x \notin A$ là một phần mở rộng của $A \in I$ nếu có thể bổ sung x vào A trong khi vẫn bảo toàn tính độc lập; nghĩa là, x là một phần mở rộng của A nếu $A \cup \{x\} \in \mathcal{I}$. Để lấy ví dụ, hãy xét một tạng ma trận đồ họa M_G. Nếu A là một tập hợp độc lập các cạnh, thì cạnh e là một phần mở rộng của A nếu và chỉ nếu e không nằm trong A và việc bổ sung x vào A không tạo một chu trình.

Nếu A là một tập hợp con độc lập trong một tạng ma trận M, ta nói rằng A là cực đại nếu nó không có các phần mở rộng. Nghĩa là, A là cực đại nếu nó không nằm trong bất kỳ tập hợp con độc lập nào lớn hơn của M. Tính chất dưới đây thường tỏ ra hữu ích.

Định lý 17.6

Tất cả các tập hợp con độc lập cực đại trong một tạng ma trận đều có cùng kích cỡ.

Chứng minh Giả sử ngược lại rằng A là một tập hợp con độc lập cực đại của M và ở đó tồn tại một tập hợp con B độc lập cực đại lớn hơn của M. Thì, tính chất trao đổi hàm ý A có thể mở rộng theo một tập hợp độc lập lớn hơn $A \cup \{x\}$ với một $x \in B - A$, ngược lại với giả thiết rằng A là cực đại.

Để minh họa cho định lý này, ta xét một tạng ma trận đồ họa M_G của một đồ thị liên thông, không có hướng, G. Mọi tập hợp con độc lập cực đại của M_G phải là một cây tự do có chính xác $|V| - 1$ cạnh liên thông với tất cả các đỉnh của G. Kiểu cây như vậy được gọi là một ***cây tỏa nhánh*** [spanning tree] của G.

Ta nói rằng một tạng ma trận $M = (S, \mathcal{I})$ được ***gia trọng*** [weighted] nếu có một hàm trọng số kết hợp w gán một trọng số dương ngặt $w(x)$ cho mỗi thành phần $x \in S$. Hàm trọng số w mở rộng theo các tập hợp con của S bằng phép lấy tổng:

$$w(A) = \sum_{x \in A} w(x)$$

với bất kỳ $A \subseteq S$. Ví dụ, nếu cho $w(e)$ thể hiện chiều dài của một cạnh e trong một tạng ma trận đồ họa M_G, thì $w(A)$ là tổng chiều dài của các cạnh trong tập hợp cạnh A.

17.4.2 Các thuật toán tham trên một tạng ma trận gia trọng

Nhiều bài toán mà cách tiếp cận tham cung cấp các giải pháp tối ưu có thể được diễn đạt chính xác theo dạng tìm một tập hợp con độc lập có trọng số cực đại trong một tạng ma trận gia trọng. Nghĩa là, ta có một

tạng ma trận gia trọng $M = (S, \mathcal{I})$, và muốn tìm một tập hợp độc lập $A \in \mathcal{I}$ sao cho w (A) được cực đại hóa. Ta gọi một tập hợp con độc lập và có trọng số khả dĩ cực đại như vậy là một tập hợp con ***tối ưu*** của tạng ma trận. Bởi trọng số $w(x)$ của bất kỳ thành phần $x \in S$ là dương, nên một tập hợp con tối ưu luôn là một tập hợp con độc lập cực đại—nó luôn giúp tạo cho A càng lớn càng tốt.

Ví dụ, trong ***bài toán cây tỏa nhánh cực tiểu***, ta có một đồ thị liên thông không có hướng $G = (V, E)$ và một hàm chiều dài w sao cho $w(e)$ là chiều dài (dương) của cạnh e. (Ta dùng thuật ngữ "chiều dài" ở đây để chỉ các trọng số cạnh ban đầu của đồ thị, dành riêng thuật ngữ "trọng số" để chỉ các trọng số trong tạng ma trận kết hợp.) Ta được yêu cầu tìm một tập hợp con các cạnh liên thông tất cả các đỉnh với nhau và có tổng chiều dài tối thiểu. Để xem nó như một bài toán tìm một tập hợp con tối ưu của một tạng ma trận, ta xét tạng ma trận gia trọng M_G với hàm trọng số w', ở đó $w'(e) = w_0 - w(e)$ và w_0 lớn hơn chiều dài cực đại của bất kỳ cạnh nào. Trong tạng ma trận gia trọng này, tất cả các trọng số đều là dương và một tập hợp con tối ưu là một cây tỏa nhánh có tổng chiều dài cực tiểu trong đồ thị ban đầu. Cụ thể hơn, mỗi tập hợp con độc lập cực đại A tương ứng với một cây tỏa nhánh, và bởi

$$w'(A) = (|V| - 1)w_0 - w(A)$$

với bất kỳ tập hợp con A độc lập cực đại nào, tập hợp con độc lập tối đa hóa $w'(A)$ phải giảm thiểu $w(A)$. Như vậy, bất kỳ thuật toán nào có thể tìm ra một tập hợp con tối ưu A trong một tạng ma trận tùy ý đều có thể giải quyết bài toán cây tỏa nhánh cực tiểu.

Chương 24 có nêu các thuật toán cho bài toán cây tỏa nhánh cực tiểu, nhưng ở đây ta đưa ra một thuật toán tham làm việc với bất kỳ tạng ma trận gia trọng nào. Thuật toán chấp nhận dưới dạng nhập liệu một tạng ma trận gia trọng $M = (S, \mathcal{I})$ với một hàm trọng số dương kết hợp w, và trả về một tập hợp con tối ưu A. Trong mã giả, ta thể hiện các thành phần của M theo $S[M]$ và $\mathcal{I}[M]$ và hàm trọng số theo w. Thuật toán là "tham" bởi nó lần lượt xét từng thành phần $x \in S$ theo thứ tự của trọng số không tăng và lập tức bổ sung nó vào tập hợp A đang được tích lũy nếu $A \cup \{x\}$ là độc lập.

GREEDY(M, w)

$A \leftarrow \emptyset$

sort $S[M]$ into nonincreasing order by trọng số w

for mỗi $x \in S[M]$, taken trong nonincreasing order by trọng số $w(x)$

    do if $A \cup \{x\} \in \mathcal{I}[M]$

5 **then** $A \leftarrow A \cup \{x\}$

6 **return** A

Các thành phần của S được xét lần lượt, theo thứ tự của trọng số không tăng. Nếu thành phần x đang được xét có thể bổ sung vào A trong khi vẫn duy trì tính độc lập của A, thì đúng là nó. Bằng không, x được gạt bỏ. Bởi tập hợp trống là độc lập theo định nghĩa của một tạng ma trận, và bởi x chỉ được bổ sung vào A nếu $A \cup \{x\}$ là độc lập, nên tập hợp con A luôn độc lập, theo phương pháp quy nạp. Do đó, GREEDY luôn trả về một tập hợp con độc lập A. Như sẽ thấy dưới đây, A là một tập hợp con có trọng số khả dĩ cực đại, sao cho A là một tập hợp con tối ưu.

Thời gian thực hiện của GREEDY cũng dễ phân tích. Cho n thể hiện $|S|$. Giai đoạn sắp xếp của GREEDY mất một thời gian $O(n \lg n)$. Dòng 4 được thi hành chính xác n lần, mỗi lần cho từng thành phần của S. Mỗi lần thi hành dòng 4 yêu cầu một đợt kiểm tra xem tập hợp $A \cup \{x\}$ có độc lập hay không. Nếu mỗi đợt kiểm tra như vậy mất một thời gian $O(f(n))$, nguyên cả thuật toán chạy trong thời gian $O(n \lg n + n f(n))$.

Giờ đây ta chứng minh GREEDY trả về một tập hợp con tối ưu.

***Bổ đề* 17.7 (*Các tạng ma trận biểu lộ tính chất của sự chọn lựa tham*)**

Giả sử $M = (S, \mathcal{I})$ là một tạng ma trận gia trọng với hàm trọng số w và rằng S được sắp xếp theo thứ tự không tăng theo trọng số. Cho x là thành phần đầu tiên của S sao cho $\{x\}$ là độc lập, nếu có bất kỳ x nào như vậy tồn tại. Nếu x tồn tại, thì ở đó tồn tại một tập hợp con tối ưu A của S chứa x.

Chứng minh Nếu không có x nào như vậy tồn tại, thì tập hợp con độc lập duy nhất là tập hợp trống và coi như xong. Bằng không, cho B là một tập hợp con tối ưu không trống bất kỳ. Giả sử $x \notin B$; bằng không, ta cho $A = B$ và coi như xong.

Không có thành phần nào của B có trọng số lớn hơn $w(x)$. Để xem điều này, ta nhận thấy $y \in B$ hàm ý $\{y\}$ là độc lập, bởi $B \in \mathcal{I}$ và $\mathcal{I}$ là di truyền. Do đó, chọn lựa x của chúng ta bảo đảm rằng $w(x) \geq w(y)$ với bất kỳ $y \in B$.

Kiến tạo tập hợp A như sau. Bắt đầu với $A = \{x\}$. Nhờ chọn lựa x, A là độc lập. Dùng tính chất trao đổi, liên tục tìm một thành phần mới của B có thể bổ sung vào A cho đến khi $|A| = |B|$ trong khi vẫn bảo toàn sự độc lập của A. Sau đó, $A = B - \{y\} \cup \{x\}$ với một $y \in B$, và như vậy

$$\begin{aligned} w(A) &= w(B) - w(y) + w(x) \\ &\geq w(B). \end{aligned}$$

Bởi B là tối ưu, nên A cũng phải tối ưu, và bởi $x \in A$, nên bổ đề được chứng minh.

Sau đó, ta chứng tỏ nếu ban đầu một thành phần không phải là một tùy chọn, thì nó không thể là một tùy chọn về sau.

Bổ đề 17.8

Cho $M = (S, \mathcal{I})$ là một tạng ma trận bất kỳ. Nếu x là một thành phần của S sao cho x không phải là một phần mở rộng của $\emptyset$, thì x không phải là một phần mở rộng của bất kỳ tập hợp con độc lập A nào của S.

Chứng minh Ta chứng minh qua sự mâu thuẫn. Giả sử x là một phần mở rộng của A nhưng không thuộc $\emptyset$. Bởi x là một phần mở rộng của A, ta có $A \cup \{x\}$ là độc lập. Bởi $\mathcal{I}$ là di truyền, $\{x\}$ phải là độc lập, điều này mâu thuẫn với giả thiết rằng x không phải là một phần mở rộng của $\emptyset$.

Bổ đề 17.8 cho biết mọi thành phần không thể dùng được ngay có thể sẽ không bao giờ được dùng. Do đó, GREEDY không thể thực hiện một lỗi bằng cách bỏ qua bất kỳ thành phần ban đầu nào trong S không phải là một phần mở rộng của $\emptyset$, bởi chúng có thể không bao giờ được dùng.

***Bổ đề* 17.9 (*Các tạng ma trận biểu lộ tính chất cấu trúc con tối ưu*)**

Cho x là thành phần đầu tiên của S được GREEDY chọn cho tạng ma trận gia trọng $M = (S, \mathcal{I})$. Bài toán còn lại để tìm một tập hợp con độc lập trọng số cực đại chứa x sẽ thu giảm vào việc tìm một tập hợp con độc lập trọng số cực đại của tạng ma trận gia trọng $M' = (S', \mathcal{I}')$, ở đó

$S' = \{y \in S : \{x, y\} \in I\}$,

$I' = \{B \subseteq S - \{x\} : B \cup \{x\} \in I\}$, và

hàm trọng số của M' là hàm trọng số của M, được hạn chế vào S'. (Ta gọi M' là ***dạng rút gọn*** của M theo thành phần x.)

Chứng minh Nếu A là một tập hợp con độc lập trọng số cực đại bất kỳ của M chứa x, thì $A' = A - \{x\}$ là một tập hợp con độc lập của M'. Ngược lại, một tập hợp con độc lập A' bất kỳ của M' cho ra một tập hợp con độc lập $A = A' \cup \{x\}$ của M. Bởi trong cả hai trường hợp ta có $w(A) = w(A')+w(x)$, một giải pháp trọng số cực đại trong M chứa x cho ra một giải pháp trọng số cực đại trong M', và ngược lại.

***Định lý* 17.10 (*Tính đúng đắn của thuật toán tham trên các tạng ma trận*)**

Nếu $M = (S, \mathcal{I})$ là một tạng ma trận gia trọng với hàm trọng số w, thì lệnh gọi GREEDY(M, w) trả về một tập hợp con tối ưu.

Chứng minh Theo Bổ đề 17.8, bất kỳ thành phần nào được bỏ qua lúc đầu vì chúng không phải là các phần mở rộng của $\emptyset$ đều có thể quên đi, bởi chúng có thể chẳng bao giờ hữu ích. Một khi thành phần đầu tiên x được chọn, Bổ đề 17.7 hàm ý GREEDY không phạm lỗi bằng cách cộng x vào A, bởi ở đó tồn tại một tập hợp con tối ưu chứa x. Cuối cùng, Bổ đề 17.9 hàm ý bài toán còn lại đó là tìm một tập hợp con tối ưu trong tạng ma trận M' là dạng rút gọn của M theo x. Sau khi thủ tục GREEDY ấn định A theo $\{x\}$, tất cả các bước còn lại đều có thể được diễn dịch là tác động trong tạng ma trận $M' = (S', \mathcal{I}')$, bởi vì B độc lập trong M' nếu và chỉ nếu $B \cup \{x\}$ độc lập trong M, với tất cả các tập hợp $B \in \mathcal{I}'$. Như vậy, phép toán tiếp theo của GREEDY sẽ tìm một tập hợp con độc lập trọng số cực đại cho M', và phép toán chung của GREEDY sẽ tìm một tập hợp con độc lập trọng số cực đại cho M.

Bài tập

17.4-1

Chứng tỏ $(S, \mathcal{I}_k)$ là một tạng ma trận, ở đó S là một tập hợp hữu hạn bất kỳ và $\mathcal{I}_k$ là tập hợp tất cả các tập hợp con của S có kích cỡ tối đa k, ở đó $k \leq |S|$.

17.4-2 *

Cho một ma trận T có giá trị thực $n \times n$, chứng tỏ (S, I) là một tạng ma trận, ở đó S là tập hợp các cột của T và $A \in \mathcal{I}$ nếu và chỉ nếu các cột trong A độc lập theo tuyến tính.

17.4-3 *

Chứng tỏ nếu (S,I) là một tạng ma trận, thì $(S, \mathcal{I}')$ là một tạng ma trận, ở đó $\mathcal{I}' = \{A' : S - A'$ chứa một $A \in T$ cực đại$\}$. Nghĩa là, các tập hợp độc lập cực đại của $(S', \mathcal{I}')$ chỉ là các số bù của các tập hợp độc lập cực đại của $(S, \mathcal{I})$.

17.4-4

Cho S là một tập hợp hữu hạn và cho $S_1, S_2, ..., S_k$ là một phân hoạch của S thành các tập hợp con rời nhau không trống. Hãy định nghĩa cấu trúc (S,I) theo điều kiện là $\mathcal{I} = \{A : |A \cap S_i| \leq 1$ với $i = 1, 2, ..., k\}$. Chứng tỏ $(S,\mathcal{I})$ là một tạng ma trận. Nghĩa là, tập hợp của tất cả các tập hợp A chứa tối đa một phần tử trong mỗi khối của phân hoạch sẽ xác định các tập hợp độc lập của một tạng ma trận.

17.4-5

Nêu cách biến đổi hàm trọng số của một bài toán tạng ma trận gia trọng, ở đó giải pháp tối ưu mong muốn là một tập hợp con độc lập cực đại có *trọng số cực tiểu*, để khiến nó thành một bài toán tạng ma trận gia trọng chuẩn. Chứng tỏ cẩn thận phép biến đổi của bạn là đúng đắn.

17.5 Bài toán lên lịch công việc

Một bài toán thú vị có thể được giải quyết bằng các tạng ma trận đó là bài toán lên lịch các công việc thời gian đơn vị một cách tối ưu trên một bộ xử lý đơn lẻ, ở đó mỗi công việc có một hạn chót và một hình phạt phải được thanh toán nếu hạn chót bị bỏ lỡ.

Bài toán có vẻ phức tạp, nhưng nó có thể được giải quyết thật đơn giản bằng một thuật toán tham.

Một ***công việc thời gian đơn vị*** [unit-time task] là một công việc, chẳng hạn như một chương trình chạy trên một máy tính, yêu cầu chính xác một đơn vị thời gian để hoàn thành. Cho một tập hợp hữu hạn S gồm các công việc thời gian đơn vị, một ***lịch biểu*** cho S là một phép hoán vị của S chỉ định thứ tự ở đó các công việc này phải được thực hiện. Công việc đầu tiên trong lịch biểu bắt đầu tại thời gian 0 và hoàn tất tại thời gian 1, công việc thứ hai bắt đầu tại thời gian 1 và hoàn tất tại thời gian 2, vân vân.

Bài toán ***lên lịch các công việc thời gian đơn vị với các hạn chót và các hình phạt***

cho một bộ xử lý đơn lẻ có các nhập liệu sau đây:

- một tập hợp $S = \{1, 2, ..., n\}$ gồm n công việc thời gian đơn vị;
- một tập hợp ***n hạn chót*** số nguyên $d_1, d_2, ..., d_n$, sao cho mỗi d_i thỏa $1 \leq d_i \leq n$ và công việc i được giả sử hoàn tất theo thời gian d_i; và
- một tập hợp n trọng số không âm hoặc ***các hình phạt*** $w_1, w_2, ..., w_n$, sao cho một hình phạt w_i phải được gánh chịu nếu công việc i không hoàn tất theo thời gian d_i và không phải gánh chịu hình phạt nào nếu một công việc hoàn tất theo hạn chót của nó.

Ta được yêu cầu tìm ra một lịch biểu cho S giảm thiểu tổng hình phạt bị gánh chịu đối với các hạn chót bỏ lỡ.

Xét một lịch biểu đã cho. Ta nói rằng một công việc là ***trễ*** trong lịch biểu này nếu nó hoàn tất sau thời hạn chót của nó. Bằng không, công việc là ***sớm*** trong lịch biểu. Một lịch biểu tùy ý có thể luôn được đặt vào

biểu mẫu sớm-ưu tiên [early-first form], ở đó các công việc sớm đứng trước các công việc trễ. Để xem điều này, bạn lưu ý nếu một công việc sớm x theo sau một công việc trễ y, thì ta có thể chuyển đổi các vị trí x và y mà không làm ảnh hưởng đến x đang sớm hoặc y đang trễ.

Cũng vậy, ta biện luận rằng lúc nào cũng có thể đặt một lịch biểu tùy ý vào ***biểu mẫu chính tắc,*** ở đó các công việc sớm đứng trước các công việc trễ và các công việc sớm được lên lịch theo thứ tự của các hạn chót không giảm. Để thực hiện, ta đặt lịch biểu vào biểu mẫu sớm-đầu tiên. Như vậy, miễn là có hai công việc sớm i và j hoàn tất tại các thời gian tương ứng k và $k + 1$ trong lịch biểu sao cho $d_j < d_i$, ta tráo các vị trí của i và j. Bởi công việc j là sớm trước khi tráo, nên $k + 1 \leq d_j$. Do đó, $k + 1 < d_i$, và do đó công việc i vẫn sớm sau khi tráo. Công việc j được dời sớm hơn trong lịch biểu, do đó nó cũng vẫn sớm sau khi tráo.

Như vậy đợt tìm kiếm một lịch biểu tối ưu sẽ rút gọn vào việc tìm một tập hợp A gồm các công việc phải là sớm trong lịch biểu tối ưu. Sau khi A được xác định, ta có thể tạo lịch biểu thực tế bằng cách liệt kê các thành phần của A theo thứ tự hạn chót không giảm, sau đó liệt kê các công việc trễ (tức, $S - A$) theo một thứ tự bất kỳ, tạo ra một kiểu sắp xếp thứ tự chính tắc của lịch biểu tối ưu.

Ta nói rằng một tập hợp A các công việc là ***độc lập*** nếu ở đó tồn tại một lịch biểu cho các công việc này sao cho không có công việc nào bị trễ. Rõ ràng, tập hợp các công việc sớm cho một lịch biểu sẽ hình thành một tập hợp độc lập các công việc. Cho $\mathcal{I}$ thể hiện tập hợp tất cả các tập hợp độc lập của các công việc.

Xét bài toán xác định một tập hợp A các công việc có độc lập hay không. Với $t = 1, 2, \ldots, n$, cho $N_t(A)$ thể hiện số lượng các công việc trong A có hạn chót là t hoặc sớm hơn.

Bổ đề 17.11

Với bất kỳ tập hợp các công việc A nào, các phát biểu dưới đây là tương đương.

1. Tập hợp A là độc lập.

2. Với $t = 1, 2, \ldots, n$, ta có $N_t(A) \leq t$.

3. Nếu các công việc trong A được lên lịch theo thứ tự của các hạn chót không giảm, thì không có công việc nào bị trễ.

Chứng minh Rõ ràng, nếu $N_t(A) > t$ với một t, thì không có cách nào để tạo một lịch biểu không có công việc trễ nào cho tập hợp A, bởi có

trên t công việc để hoàn tất trước thời gian t. Do đó, (1) hàm ý (2). Nếu (2) vẫn còn, thì (3) phải theo: không có cách nào để "bị kẹt" khi lên lịch các công việc theo thứ tự các hạn chót không giảm, bởi (2) hàm ý rằng hạn chót lớn nhất thứ i tối đa là i. Cuối cùng, (3) hàm ý (1) một cách không đáng kể.

Dùng tính chất 2 của Bổ đề 17.11, ta có thể dễ dàng tính toán xem một tập hợp các công việc đã cho có độc lập hay không (xem Bài tập 17.5-2).

Bài toán giảm thiểu tổng các hình phạt của các công việc trễ cũng tương tự như bài toán tối đa hóa tổng các hình phạt của các công việc sớm. Như vậy, định lý dưới đây bảo đảm ta có thể dùng thuật toán tham để tìm ra một tập hợp độc lập A gồm các công việc có tổng hình phạt cực đại.

Định lý 17.12

Nếu S là một tập hợp các công việc thời gian đơn vị có các hạn chót, và $\mathcal{I}$ là tập hợp tất cả các tập hợp độc lập các công việc, thì hệ thống tương ứng $(S, \mathcal{I})$ là một tạng ma trận.

Chứng minh Mọi tập hợp con của một tập hợp độc lập các công việc chắc chắn là độc lập. Để chứng minh tính chất trao đổi, ta giả sử rằng B và A là các tập hợp độc lập các công việc và rằng $|B| > |A|$. Cho k là t lớn nhất sao cho $N_t(B) \leq N_t(A)$. Bởi $N_n(B) = |B|$ và $N_n(A) = |A|$, nhưng $|B| > |A|$, ta phải có $k < n$ và $N_j(B) > N_j(A)$ với tất cả j trong miền giá trị $k + 1 \leq j \leq n$. Do đó, B chứa các công việc có hạn chót $k + 1$ nhiều hơn A. Cho x là một công việc trong $B - A$ có hạn chót $k + 1$. Cho $A' = A \cup \{x\}$.

Giờ đây, ta chứng tỏ A' phải là độc lập nhờ dùng tính chất 2 của Bổ đề 17.11. Với $1 \leq t \leq k$, ta có $N_t(A') = N_t(A) \leq t$, bởi A là độc lập. Với $k < t \leq n$, ta có $N_t(A') \leq N_t(B) \leq t$, bởi B là độc lập. Do đó, A' là độc lập, hoàn tất phần chứng minh của chúng ta rằng $(S, \mathcal{I})$ là một tạng ma trận.

Theo Định lý 17.10, ta có thể dùng một thuật toán tham để tìm ra một tập hợp độc lập trọng số cực đại gồm các công việc A. Sau đó, ta có thể tạo một lịch biểu tối ưu có các công việc trong A làm các công việc sớm của nó. Phương pháp này là một thuật toán hiệu quả để lên lịch các công việc thời gian đơn vị có các hạn chót và các hình phạt cho một bộ xử lý đơn lẻ. Thời gian thực hiện là $O(n^2)$ nhờ dùng GREEDY, bởi mỗi trong số $O(n)$ đợt kiểm tra độc lập mà thuật toán đó tạo sẽ mất một thời gian $O(n)$ (xem Bài tập 17.5-2). Ta có một thực thi

nhanh hơn trong Bài toán 17-3.

	Công việc						
	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

Hình 17.7 Một minh dụ của bài toán lên lịch các công việc thời gian đơn vị có các hạn chót và các hình phạt cho một bộ xử lý đơn lẻ.

Hình 17.7 cho ta một ví dụ về một bài toán lên lịch các công việc thời gian đơn vị có các hạn chót và các hình phạt cho một bộ xử lý đơn lẻ. Trong ví dụ này, thuật toán tham lựa các công việc 1, 2, 3, và 4, sau đó loại bỏ các công việc 5 và 6, và cuối cùng chấp nhận công việc 7. Lịch biểu tối ưu cuối cùng là

$<2, 4, 1, 3, 7, 5, 6>$,

has một rổng hình phạt phải gánh là $w_5 + w_6 = 50$.

Bài tập

17.5-1

Giải trường hợp của bài toán lên lịch đã cho trong Hình 17.7, nhưng với mỗi hình phạt w_i được thay bởi 80 - w_i.

17.5-2

Nêu cách sử dụng tính chất 2 của Bổ đề 17.11 để xác định trong thời gian $O(|A|)$ một tập hợp các công việc A đã cho có độc lập hay không.

Các Bài Toán

17-1 Thay đồng xu

Xét bài toán thay đổi cho n cent dùng số lượng đồng xu ít nhất.

a. Mô tả một thuật toán tham để thay đổi bao gồm các đồng 25 xu, đồng hào, niken, và peni. Chứng minh thuật toán của bạn cho ra một giải pháp tối ưu.

b. Giả sử các đồng xu sẵn có nằm trong các loại đơn vị tiền c^0, c^1,..., c^k với một số nguyên $c > 1$ và $k \geq 1$. Chứng tỏ thuật toán tham luôn cho ra một giải pháp tối ưu.

c. Nêu một tập hợp các loại đơn vị đồng xu mà thuật toán tham không mang lại một tối ưu giải pháp.

17-2 Các đồ thị con phi chu trình

a. Cho $G = (V, E)$ là một đồ thị không có hướng. Dùng phần định nghĩa của một tạng ma trận, chứng tỏ $(E, \mathcal{I})$ *là* một tạng ma trận, ở đó $A \in \mathcal{I}$ nếu và chỉ nếu *A* là một tập hợp con phi chu trình của *E*.

b. Ma trận liên thuộc của một đồ thị không có hướng $G = (V, E)$ là một ma trận $M \times |V| \times |E|$ sao cho $M_{ve} = 1$ nếu cạnh *e* là liên thuộc trên đỉnh *v*, và bằng không $M_{ve} = 0$. Chứng tỏ một tập hợp các cột của *M* là độc lập theo tuyến tính nếu và chỉ nếu tập hợp các cạnh tương ứng là phi chu trình. Sau đó, dùng kết quả của Bài tập 17.4-2 để cung cấp một chứng minh khác rằng $(E, \mathcal{I})$ của phần (a) là tạng ma trận.

c. Giả sử một trọng số không âm $w(e)$ kết hợp với mỗi cạnh trong một đồ thị không có hướng $G = (V, E)$. Nêu một thuật toán hiệu quả để tìm một tập hợp con phi chu trình của *E* có tổng trọng số cực đại.

d. Cho $G(V,E)$ là một đồ thị có hướng tùy ý, và cho $(E, \mathcal{I})$ được định nghĩa sao cho $A \in I$ nếu và chỉ nếu *A* không chứa bất kỳ chu trình có hướng nào. Nêu một ví dụ của một đồ thị có hướng *G* sao cho hệ thống kết hợp (E, I) không phải là một tạng ma trận. Chỉ định điều kiện định nghĩa nào của một tạng ma trận không đứng vững.

e. Ma trận liên thuộc của một đồ thị có hướng $G = (V, E)$ là một ma trận $|V| \times |E|$ sao cho $M_{ve} = -1$ nếu cạnh *e* rời đỉnh *v*, $M_{ve} = 1$ nếu cạnh *e* nhập đỉnh *v*, và bằng không $M_{ve} = 0$. Chứng tỏ nếu một tập hợp các cạnh của *G* độc lập theo tuyến tính, thì tập hợp tương ứng các cạnh sẽ không chứa một chu trình có hướng.

f. Bài tập 17.4-2 cho ta biết tập hợp các tập hợp độc lập theo tuyến tính gồm các cột của bất kỳ ma trận *M* nào sẽ hình thành một tạng ma trận. Giải thích cẩn thận tại sao các kết quả của các phần (d) và (e) không mâu thuẫn. Không thể có sự tương ứng hoàn hảo tới mức nào giữa khái niệm về một tập hợp các cạnh là phi chu trình với khái niệm về một tập hợp kết hợp gồm các cột của ma trận liên thuộc là độc lập theo tuyến tính?

17-3 Các biến thể về lên lịch

Xét thuật toán dưới đây để giải quyết bài toán trong Đoạn 17.5 về cách lên lịch các công việc thời gian đơn vị có các hạn chót và các hình

phạt. Cho tất cả các *n* khe thời gian trống từ đầu, ở đó khe thời gian *i* là khe chiều dài đơn vị của thời gian hoàn tất tại thời gian *i*. Ta xét các công việc theo thứ tự của hình phạt giảm đơn điệu. Khi xét công việc *j*, nếu ở đó tồn tại một khe thời gian tại hoặc trước hạn chót d_j của *j* vẫn còn trống, gán công việc *j* cho khe mới nhất như vậy, điền nó. Nếu không có khe nào như vậy, bạn gán công việc *j* cho khe mới nhất của các khe vẫn chưa điền.

a. Chứng tỏ thuật toán này luôn cho ra một đáp án tối ưu.

b. Dùng rừng tập hợp rời [disjoint-set] nhanh nêu trong Đoạn 22.3 để thực thi thuật toán một cách hiệu quả. Giả sử tập hợp các công việc nhập liệu đã được sắp xếp thứ tự giảm đơn điệu theo hình phạt. Phân tích thời gian thực hiện của tiến trình thực thi.

Ghi chú Chương

Có thể tham khảo thêm về các thuật toán tham và các tạng ma trận trong Lawler [132] và Papadimitriou và Steiglitz [154].

Thuật toán tham xuất hiện lần đầu tiên trong tài liệu giáo khoa về tối ưu hóa tổ hợp trong một bài viết 1971 của Edmonds [62], tuy vậy lý thuyết về các tạng ma trận đã có từ thời bài viết 1935 của Whitney [200].

Chứng minh của chúng ta về tính đúng đắn của thuật toán tham cho bài toán lựa chọn hoạt động dựa trên chứng minh của Gavril [80]. Bài toán lên lịch công việc được nghiên cứu trong Lawler [132], Horowitz và Sahni [105], Brassard và Bratley [33].

Các mã Huffman đã được sáng chế vào năm 1952 [107]; Lelewer và Hirschberg [136) nghiên cứu các kỹ thuật nén dữ liệu được xem là từ năm 1987.

Một phần mở rộng của lý thuyết tạng ma trận sang lý thuyết tạng tham [greedoid] đã được khai phá bởi Korte và Lovasz [127, 128, 129, 130], họ đã tổng quát hóa rất nhiều lý thuyết trình bày ở đây.

18 Phân Tích Khấu Trừ

Trong phép ***phân tích khấu trừ,*** thời gian yêu cầu để thực hiện một dãy các phép toán cấu trúc dữ liệu được tính bình quân trên tất cả các phép toán được thực hiện. Có thể dùng phép phân tích khấu trừ để chứng tỏ mức hao phí trung bình của một phép toán là nhỏ, nếu như ta lấy bình quân trên một dãy các phép toán, cho dù một phép toán đơn lẻ có thể là tốn kém. Phân tích khấu trừ khác với phân tích trường hợp trung bình ở chỗ không có liên quan đến xác suất; phân tích khấu trừ bảo đảm *khả năng thực hiện trung bình của mỗi phép toán trong trường hợp xấu nhất.*

Ba đoạn đầu tiên của chương này đề cập ba kỹ thuật thông dụng nhất được dùng trong phân tích khấu trừ. Đoạn 18.1 bắt đầu với phương pháp kết tập [aggregate], ở đó ta xác định một cận trên $T(n)$ trên tổng mức hao phí của một dãy n phép toán. Như vậy, mức hao phí khấu trừ của mỗi phép toán là $T(n)/n$.

Đoạn 18.2 đề cập phương pháp kế toán, ở đó ta xác định mức hao phí khấu trừ của mỗi phép toán. Khi có nhiều kiểu phép toán, mỗi kiểu phép toán có thể có một mức hao phí khấu trừ khác nhau. Phương pháp kế toán "tính công dư" sớm cho vài phép toán trong dãy, lưu trữ khoản tính công dư dưới dạng "tín dụng trả trước" trên các đối tượng cụ thể trong cấu trúc dữ liệu. Khoản tín dụng được dùng về sau trong dãy để thanh toán cho các phép toán được tính công nhỏ hơn mức hao phí thực tế của chúng.

Đoạn 18.3 mô tả phương pháp thế [potential method], nó cũng giống phương pháp kế toán ở chỗ ta xác định mức hao phí khấu trừ của mỗi phép toán và có thể tính công dư sớm cho các phép toán để bù cho khoản tính công thiếu về sau. Phương pháp thế duy trì khoản tín dụng dưới dạng "năng lượng thế" của cấu trúc dữ liệu thay vì kết hợp khoản tín dụng với các đối tượng riêng lẻ trong cấu trúc dữ liệu.

Ta sẽ dùng hai ví dụ để xét ba mô hình này. Một ví dụ là một ngăn xếp với phép toán bổ sung MULTIPOP, "kéo" [pops] vài đối tượng ngay lập tức. Ví dụ kia là một bộ đếm nhị phân đếm lên từ 0 thông qua phép toán đơn lẻ INCREMENT.

Khi đọc chương này, bạn nên nhớ khoản tính công được gán trong

khi phân tích khấu trừ là chỉ để phân tích mà thôi. Chúng không được xuất hiện trong mã. Ví dụ, nếu một khoản tín dụng được gán cho một đối tượng x khi dùng phương pháp kế toán, ta không cần gán một khoản thích hợp cho một thuộc tính $credit[x]$ trong mã.

Nếu hiểu thấu đáo một cấu trúc dữ liệu cụ thể nhờ thực hiện một đợt phân tích khấu trừ, bạn có thể tối ưu hóa thiết kế. Ví dụ, trong Đoạn 18.4, ta dùng phương pháp thế để phân tích một bảng mở rộng và rút giảm động.

18.1 Phương pháp kết tập

Trong ***phương pháp kết tập*** của phép phân tích khấu trừ, ta chứng tỏ với tất cả n, một dãy n phép toán chiếm tổng cộng $T(n)$ thời gian trường hợp xấu nhất. Như vậy, trong trường hợp xấu nhất, mức hao phí trung bình, hoặc ***mức hao phí khấu trừ***, của mỗi phép toán là $T(n)/n$. Lưu ý, mức hao phí khấu trừ này áp dụng cho mỗi phép toán, thậm chí khi có vài kiểu phép toán trong dãy. Phương pháp kế toán và phương pháp thế, sẽ được giải thích sau trong chương này, có thể gán các mức hao phí khấu trừ khác nhau cho các kiểu phép toán khác nhau.

Các phép toán ngăn xếp

Trong ví dụ đầu tiên của chúng ta về phương pháp kết tập, ta phân tích các ngăn xếp đã được tăng cường với một phép toán mới. Đoạn 11.1 đã trình bày hai phép toán ngăn xếp căn bản, mỗi phép kéo dài $O(1)$ thời gian:

PUSH(S, x) đẩy đối tượng x lên trên ngăn xếp S.

POP(S) kéo đầu của ngăn xếp S ra và trả về đối tượng được kéo ra.

Bởi từng phép toán này chạy trong $O(1)$ thời gian, ta hãy xem mức hao phí của mỗi phép toán là 1. Do đó, tổng mức hao phí của một dãy n phép toán PUSH và POP là n, và như vậy thời gian thực hiện thực tế cho n phép toán là $\Theta(n)$.

Tình huống trở thành thú vị hơn nếu ta bổ sung phép toán ngăn xếp MULTIPOP(S, k) để gỡ bỏ k đối tượng trên cùng của ngăn xếp S, hoặc kéo nguyên cả ngăn xếp nếu nó chứa ít hơn k đối tượng. Trong mã giả dưới đây, phép toán STACK-EMPTY trả về TRUE nếu không có đối tượng nào hiện nằm trên ngăn xếp, và bằng không là FALSE.

MULTIPOP(S, k)

1 **while** not STACK-EMPTY(S) và $k \neq 0$

2 **do** POP(S)

3 $k \leftarrow k - 1$

Hình 18.1 nêu a ví dụ của MULTIPOP

Đâu là thời gian thực hiện của MULTIPOP(S, k) trên một ngăn xếp gồm s đối tượng? Thời gian thực hiện thực tế là tuyến tính trong số lượng phép toán POP thực tế được thi hành, và như vậy nó đủ để phân tích MULTIPOP theo dạng các mức hao phí trừu tượng là 1 cho PUSH và POP. Số lượng lần lặp lại của vòng lặp **while** là số min(s, k) các đối tượng được kéo ra khỏi ngăn xếp. Với mỗi lần lặp lại vòng lặp, một lệnh gọi được thực hiện cho POP trong dòng 2. Như vậy, tổng mức hao phí của MULTIPOP là min(s, k), và thời gian thực hiện thực tế là một hàm tuyến tính của mức hao phí này.

	(a)		(b)	(c)
top→	23			
	17			
	6			
	39			
	10	top→	10	
	47			

Hình 18.1 Hành động MULTIPOP trên một ngăn xếp S, thoạt đầu nêu trong **(a)**. 4 đối tượng đầu được kéo ra bởi MULTIPOP(S, 4), mà kết quả của nó được nêu trong **(b)**. Phép toán kế tiếp là MULTIPOP(S, 7), sẽ xả trống ngăn xếp—được nêu trong **(c)**—bởi chỉ còn lại ít hơn 7 đối tượng.

Ta hãy phân tích một dãy n phép toán PUSH, POP, và MULTIPOP trên một ngăn xếp trống từ đầu. Mức hao phí trường hợp xấu nhất của một phép toán MULTIPOP trong dãy là $O(n)$, bởi kích cỡ ngăn xếp tối đa là n. Do đó, thời gian trường hợp xấu nhất của bất kỳ phép toán ngăn xếp nào sẽ là $O(n)$, và như vậy một dãy n phép toán sẽ có mức hao phí là $O(n^2)$, bởi ta có thể có $O(n)$ phép toán MULTIPOP, mỗi phép có mức hao phí là $O(n)$. Mặc dù kiểu phân tích này là đúng đắn, kết quả $O(n^2)$, có được nhờ xét mức hao phí trường hợp xấu nhất của mỗi phép toán riêng lẻ, là không chặt.

Dùng phương pháp kết tập của phép phân tích khấu trừ, ta có thể được một cận trên tốt hơn xem xét nguyên cả dãy n phép toán. Thực vậy, mặc dù một phép toán MULTIPOP đơn lẻ có thể tốn kém, bất kỳ dãy n phép toán PUSH, POP, và MULTIPOP nào trên một ngăn xếp trống từ đầu đều có thể có mức hao phí tối đa $O(n)$. Tại sao? Mỗi đối tượng có thể được kéo ra tối đa một lần cho mỗi lần nó được đẩy. Do

đó, số lần mà POP có thể được gọi trên một ngăn xếp không trống, kể cả các lần gọi trong MULTIPOP, tối đa là số lượng các phép toán PUSH, mà các phép toán này tối đa là n. Với bất kỳ giá trị nào của n, một dãy n phép toán PUSH, POP, và MULTIPOP bất kỳ đều chiếm một tổng $O(n)$ thời gian. Mức hao phí khấu trừ của một phép toán là số trung bình: $O(n)/n = O(1)$.

Xin nhắc lại rằng mặc dù ta vừa chứng tỏ mức hao phí trung bình, và do đó thời gian thực hiện, của một phép toán ngăn xếp là $O(1)$, song nó không dính dáng gì đến biện luận xác suất. Thực tế ta đã nêu một cận *ca xấu nhất* của $O(n)$ trên một dãy n phép toán. Chia tổng mức hao phí này cho n đã cho ra mức hao phí trung bình của mỗi phép toán, hoặc mức hao phí khấu trừ.

Gia số bộ đếm nhị phân

Để lấy một ví dụ khác về phương pháp kết tập, ta hãy xét bài toán thực thi một bộ đếm nhị phân k-bit đếm lên từ 0. Ta dùng một mảng $A[0..k - 1]$ bit, ở đó $length[A] = k$, làm bộ đếm. Một số nhị phân x được lưu trữ trong bộ đếm có bit cấp thấp nhất của nó trong $A[0]$ và bit cấp cao nhất trong $A[k - 1]$, sao cho $x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$. Thoạt đầu, $x = 0$, và như vậy $A[i] = 0$ với $i = 0, 1, ..., k - 1$. Để cộng 1 (modulo 2^k) vào giá trị trong bộ đếm, ta dùng thủ tục dưới đây.

Giá trị bộ đếm	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Tổng mức hao phí
0	0	0	0	0	0	0	0	[redacted]	0
1	0	0	0	0	0	0	[redacted]		1
2	0	0	0	0	0	0	1	[redacted]	3
3	0	0	0	0	0	[redacted]			4
4	0	0	0	0	0	1	0	[redacted]	7
5	0	0	0	0	0	1	[redacted]		8
6	0	0	0	0	0	1	1	[redacted]	10
7	0	0	0	0	[redacted]				11
8	0	0	0	0	1	0	0	[redacted]	15
9	0	0	0	0	1	0	[redacted]		16
10	0	0	0	0	1	0	1	[redacted]	18
11	0	0	0	0	1	[redacted]			19
12	0	0	0	0	1	1	0	[redacted]	22
13	0	0	0	0	1	1	[redacted]		23
14	0	0	0	0	1	1	1	[redacted]	25
15	0	0	0	[redacted]					26
16	0	0	0	1	0	0	0	[redacted]	31

Hình 18.2 Một bộ đếm nhị phân 8-bit khi giá trị của nó đi từ 0 đến 16 theo một dãy 16 phép toán INCREMENT. Các bit lật để đạt được giá trị kế tiếp được tô bóng. Mức hao phí thực hiện để lật các bit được nêu ở bên phải. Lưu ý, tổng mức hao phí không bao giờ lớn hơn hai lần tổng số phép toán INCREMENT.

```
INCREMENT(A)
1 i ← 0
2 while i < length[A] và A[i] = 1
3       do A[i] ← 0
4          i ← i + 1
5 if i < length[A]
6       then A[i] ← 1
```

Về cơ bản, thuật toán này giống như thuật toán đã được thực thi trong phần cứng bằng một bộ đếm mang sang lược [ripple-carry counter] (xem Đoạn 29.2.1). Hình 18.2 nêu nội dung xảy ra đối với một bộ đếm nhị phân khi nó được gia số 16 lần, bắt đầu với giá trị đầu tiên là 0 và kết thúc với giá trị 16. Tại đầu mỗi lần lặp lại của vòng lặp **while** trong các dòng 2-4, ta muốn cộng một 1 vào vị trí i. Nếu $A[i] = 1$, thì phép cộng 1 sẽ lật bit thành 0 tại vị trí i và cho ra một phép mang sang 1, để được cộng vào vị trí $i + 1$ trên lần lặp lại kế tiếp của vòng lặp. Bằng không, vòng lặp kết thúc, và như vậy, nếu $i < k$, ta biết $A[i] = 0$, sao cho phép cộng một 1 vào vị trí i, lật 0 thành một 1, được thực hiện trong dòng 6. Mức hao phí của mỗi phép toán INCREMENT là tuyến tính trong số lượng bit được lật.

Giống như trong ví dụ ngăn xếp, một sự phân tích nhanh cho ra một cận tuy là đúng nhưng không chặt. Một đợt thi hành INCREMENT sẽ mất một thời gian $\Theta(k)$ trong trường hợp xấu nhất, ở đó mảng A chứa tất cả các 1. Như vậy, một dãy n phép toán INCREMENT trên một bộ đếm zero từ đầu sẽ mất một thời gian $O(nk)$ trong trường hợp xấu nhất.

Ta có thể thắt chặt đợt phân tích để cho ra mức hao phí trường hợp xấu nhất của $O(n)$ với một dãy n INCREMENT bằng cách nhận xét rằng không phải tất cả các bit đều lật mỗi lần INCREMENT được gọi. Như Hình 18.2 đã nêu, $A[0]$ lật mỗi lần INCREMENT được gọi. Bit cấp cao nhất kế tiếp, $A[1]$, chỉ lật mọi lần khác: một dãy n phép toán INCREMENT trên một bộ đếm zero từ đầu sẽ khiến $A[1]$ lật $\lfloor n/2 \rfloor$ lần. Cũng vậy, bit $A[2]$ chỉ lật mọi lần thứ tư, hoặc $\lfloor n/4 \rfloor$ lần trong một dãy n INCREMENT. Nói chung, với $i = 0, 1, \ldots, \lfloor \lg n \rfloor$, bit $A[i]$ lật $\lfloor n/2^i \rfloor$ lần trong một dãy n phép toán INCREMENT trên một bộ đếm zero từ đầu. Với $i > \lfloor \lg n \rfloor$, bit $A[i]$ không hề lật gì cả. Như vậy, tổng của các lần lật trong dãy là

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n ,$$

theo phương trình (3.4). Do đó, thời gian trường hợp xấu nhất cho một dãy n phép toán INCREMENT trên một bộ đếm zero từ đầu là $O(n)$, như vậy mức hao phí khấu trừ của mỗi phép toán là $O(n)/n = O(1)$.

Bài tập

18.1-1

Nếu một phép toán MULTIPUSH được gộp vào tập hợp các phép toán ngăn xếp, thì cận $O(1)$ trên mức hao phí khấu trừ của các phép toán ngăn xếp tiếp tục đứng vững không?

18.1-2

Chứng tỏ nếu một phép toán DECREMENT được gộp trong ví dụ bộ đếm k-bit, n phép toán có thể có mức hao phí tối đa là $\Theta(nk)$ thời gian.

18.1-3

Một dãy n phép toán được thực hiện trên một cấu trúc dữ liệu. Phép toán thứ i có mức hao phí i nếu i là một lũy thừa chính xác của 2, mà bằng không là 1. Dùng một phương pháp kết tập của phân tích để xác định mức hao phí khấu trừ của mỗi phép toán.

18.2 Phương pháp kế toán

Trong ***phương pháp kế toán*** của phân tích khấu trừ, ta gán các khoản tính công khác biệt cho các phép toán khác nhau, với vài phép toán được tính công nhiều hoặc ít hơn mức hao phí thực tế của chúng. Khoản mà ta tính công cho một phép toán được gọi là ***mức hao phí khấu trừ*** của nó. Khi mức hao phí khấu trừ của phép toán vượt quá mức hao phí thực tế của nó, sự khác biệt được gán cho các đối tượng cụ thể trong cấu trúc dữ liệu dưới dạng ***khoản tín dụng***.

Về sau khoản tín dụng có thể được dùng để giúp thanh toán cho các phép toán có mức hao phí khấu trừ nhỏ hơn mức hao phí thực tế của chúng. Như vậy, ta có thể xem mức hao phí khấu trừ của một phép toán dưới dạng đang được tách giữa mức hao phí thực tế của nó với khoản tín dụng hoặc được ký gửi hoặc được dùng hết. Điều này rất khác với phương pháp kết tập, ở đó tất cả các phép toán có cùng mức hao phí khấu trừ.

Ta phải chọn các mức hao phí khấu trừ của các phép toán thật cẩn thận. Nếu muốn phân tích bằng các mức hao phí khấu trừ để chứng tỏ mức hao phí trung bình trong trường hợp xấu nhất của mỗi phép toán là

nhỏ, tổng mức hao phí khấu trừ của một dãy các phép toán phải là một cận trên trên tổng mức hao phí thực tế của dãy đó. Hơn nữa, như trong phương pháp kết tập, mối quan hệ này phải áp dụng cho tất cả các dãy phép toán. Như vậy, tổng khoản tín dụng kết hợp với cấu trúc dữ liệu phải luôn là không âm, bởi nó biểu thị cho khoản mà tổng các mức hao phí khấu trừ gánh chịu vượt quá tổng các mức hao phí thực tế gánh chịu. Nếu tổng khoản tín dụng được phép trở thành âm (kết quả của việc tính công thiếu sớm cho các phép toán với hứa hẹn hoàn trả khoản thanh toán về sau), thì tổng các mức hao phí khấu trừ gánh chịu vào thời điểm đó sẽ nằm dưới tổng các mức hao phí thực tế gánh chịu; với dãy các phép toán lên tới vào thời điểm đó, tổng mức hao phí khấu trừ sẽ không phải là một cận trên trên tổng mức hao phí thực tế. Như vậy, ta phải thận trọng tổng khoản tín dụng trong cấu trúc dữ liệu không bao giờ trở thành âm.

Các phép toán ngăn xếp

Để minh họa phương pháp kế toán của phân tích khấu trừ, ta hãy trở về ví dụ ngăn xếp. Chắc bạn còn nhớ các mức hao phí thực tế của các phép toán là

PUSH	1 ,
POP	1 ,
MULTIPOP	$\min(k, s)$,

ở đó k là đối số cung cấp cho MULTIPOP và s là kích cỡ ngăn xếp khi nó được gọi. Hãy gán các mức hao phí khấu trừ sau đây:

PUSH	2 ,
POP	0 ,
MULTIPOP	0 .

Lưu ý, mức hao phí khấu trừ của MULTIPOP là một hằng (0), trong khi đó mức hao phí thực tế lại biến đổi. Ở đây, cả ba mức hao phí khấu trừ đều là $O(1)$, mặc dù nói chung các mức hao phí khấu trừ của các phép toán đang được xem xét có thể khác nhau theo tiệm cận.

Giờ đây ta chứng tỏ có thể thanh toán cho bất kỳ dãy phép toán ngăn xếp nào bằng cách tính công các mức hao phí khấu trừ. Giả sử ta dùng một tờ đôla để biểu thị cho mỗi đơn vị của mức hao phí. Ta bắt đầu với một ngăn xếp trống. Hãy nhớ lại tính tương tự của Đoạn 11.1 giữa cấu trúc dữ liệu ngăn xếp và một ngăn xếp các đĩa trong một quán ăn tự phục vụ. Khi đẩy một đĩa lên ngăn xếp, ta dùng 1 đôla để thanh toán cho mức hao phí thực tế của lần đẩy và được để lại một khoản tín dụng 1 đôla (từ 2 đôla được tính công), mà ta đặt lên trên đĩa. Tại một điểm

bất kỳ theo thời gian, mọi đĩa trên ngăn xếp đều có một đôla khoản tín dụng trên nó.

Đôla được lưu trữ trên đĩa là khoản trả trước cho mức hao phí của tiến trình kéo nó ra từ ngăn xếp. Khi ta thi hành một phép toán POP, ta không tính công cho phép toán và thanh toán mức hao phí thực tế của nó bằng khoản tín dụng lưu trữ trong ngăn xếp. Để kéo ra một đĩa, ta lấy đôla tín dụng ra khỏi đĩa và dùng nó để thanh toán mức hao phí thực tế của phép toán. Như vậy, nhờ tính công thêm một ít cho phép toán PUSH, ta không cần tính công cho phép toán POP.

Hơn nữa, ta cũng chẳng cần tính công cho các phép toán MULTIPOP. Để kéo ra đĩa đầu tiên, ta lấy đôla của khoản tín dụng ra khỏi đĩa và dùng nó để thanh toán mức hao phí thực tế của một phép toán POP. Để kéo ra một đĩa thứ hai, ta lại có một đôla tín dụng trên đĩa để thanh toán cho phép toán POP, và cứ thế tiếp tục. Như vậy, ta luôn tính công trước ít nhất đủ để thanh toán cho các phép toán MULTIPOP. Nói cách khác, do mỗi đĩa trên ngăn xếp có 1 đôla tín dụng trên nó, và ngăn xếp luôn có một số đĩa không âm, ta bảo đảm khoản tín dụng luôn không âm. Như vậy, với bất kỳ dãy n phép toán PUSH, POP, và MULTIPOP nào, tổng mức hao phí khấu trừ vẫn là một cận trên trên tổng mức hao phí thực tế. Do tổng mức hao phí khấu trừ là $O(n)$, nên nó là tổng mức hao phí thực tế.

Gia số một bộ đếm nhị phân

Để lấy một ví dụ khác về phương pháp kế toán, ta phân tích phép toán INCREMENT trên một bộ đếm nhị phân bắt đầu tại zero. Như đã nhận xét trên đây, thời gian thực hiện của phép toán này tỷ lệ với số lượng bit được lật, mà ta sẽ dùng làm mức hao phí cho ví dụ này. Một lần nữa ta hãy dùng một tờ đôla để biểu thị cho mỗi đơn vị hao phí (tiến trình lật một bit trong ví dụ này).

Với phân tích khấu trừ, ta hãy tính công cho một mức hao phí khấu trừ 2 đôla để xác lập một bit là 1. Khi một bit được xác lập, ta dùng 1 đôla (từ 2 đôla được tính công) để thanh toán cho xác lập thực tế của bit đó, và ta đặt đôla kia trên bit đó làm khoản tín dụng. Tại bất kỳ điểm nào theo thời gian, mọi 1 trong bộ đếm đều có một đôla tín dụng trên nó, và như vậy ta chẳng cần tính công gì cả để chỉnh lại một bit thành 0; ta chỉ cần thanh toán cho tiến trình chỉnh lại bằng tờ đôla trên bit đó.

Giờ đây, mức hao phí khấu trừ của INCREMENT có thể được xác định. Mức hao phí của việc chỉnh lại các bit trong vòng lặp **while** được thanh toán bằng tờ đôla trên các bit được chỉnh lại. Tối đa một bit được xác lập, trong dòng 6 của INCREMENT, và do đó mức hao phí khấu trừ

của một phép toán INCREMENT tối đa là 2 đôla. Số lượng 1 trong bộ đếm không bao giờ âm, và như vậy khoản tín dụng luôn không âm. Do đó, với n phép toán INCREMENT, tổng mức hao phí khấu trừ là $O(n)$, định cận tổng mức hao phí thực tế.

Bài tập

18.2-1

Một dãy phép toán ngăn xếp được thực hiện trên một ngăn xếp có kích cỡ không bao giờ vượt quá k. Sau mọi phép toán k, một bản sao của nguyên cả ngăn xếp được tạo để lưu dự phòng. Hãy chứng tỏ mức hao phí của n phép toán ngăn xếp, kể cả việc chép ngăn xếp, là $O(n)$ bằng cách gán các mức hao phí khấu trừ thích hợp cho các phép toán ngăn xếp khác nhau.

18.2-2

Làm lại Bài tập 18.1-3 dùng một phương pháp kế toán của phân tích.

18.2-3

Giả sử ta muốn không những gia số một bộ đếm mà còn chỉnh lại nó thành zero (tức là, khiến tất cả các bit trong nó thành 0). Nêu cách thực thi một bộ đếm như một vector bit sao cho bất kỳ dãy n phép toán INCREMENT và RESET nào cũng đều mất một thời gian $O(n)$ trên một bộ đếm zero từ đầu. (*Mách nước:* Giữ một biến trỏ đến 1 cấp cao hơn.)

18.3 Phương pháp thế

Thay vì biểu thị công được trả trước dưới dạng khoản tín dụng lưu trữ với các đối tượng cụ thể trong cấu trúc dữ liệu, ***phương pháp thế*** của phân tích khấu trừ biểu thị công được trả trước dưới dạng "năng lượng thế," hoặc đơn giản là "thế", có thể được phóng thích để thanh toán cho các phép toán tương lai. Thế được kết hợp với cấu trúc dữ liệu dưới dạng một toàn thể thay vì với các đối tượng cụ thể trong cấu trúc dữ liệu.

Phương pháp thế làm việc như sau. Ta bắt đầu với một cấu trúc dữ liệu ban đầu D_0 ở đó n phép toán được thực hiện. Với mỗi $i = 1, 2, ..., n$, ta cho c_i là mức hao phí thực tế của phép toán thứ i và D_i là cấu trúc dữ liệu kết quả sau khi áp dụng phép toán thứ i cho cấu trúc dữ liệu D_{i-1}. Một ***hàm thế*** [potential function] (Φ ánh xạ mỗi cấu trúc dữ liệu D_i theo một số thực $\Phi(D_i)$, là ***thế*** kết hợp với cấu trúc dữ liệu D_i. ***Mức hao phí***

khấu trừ $\hat{c}_i$ của phép toán thứ *i* đối với hàm thế (Φ được định nghĩa bởi

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) . \qquad (18.1)$$

Do đó, mức hao phí khấu trừ của mỗi phép toán là mức hao phí thực tế của nó cộng với mức gia tăng trong thế do phép toán. Theo phương trình (18.1), tổng mức hao phí khấu trừ của *n* phép toán là

$$\begin{aligned}\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= \sum_{i=1}^{n} (c_i + \Phi(D_n) - \Phi(D_0) .\end{aligned} \qquad (18.2)$$

Đẳng thức thứ hai là do phương trình (3.7), bởi mức thu gọn $\Phi(D_i)$.

Nếu ta có thể định nghĩa một hàm thế Φ sao cho ($\Phi(D_n) \geq \Phi(D_0)$, thì tổng mức hao phí khấu trừ $\sum_{i=1}^{n} \hat{c}_i$ là một cận trên trên tổng mức hao phí thực tế. Trong thực tế, không phải lúc nào ta cũng biết có thể thực hiện bao nhiêu phép toán. Do đó, nếu ta yêu cầu ($\Phi(D_i) \geq \Phi(D_0)$ với tất cả *i*, thì ta bảo đảm, như trong phương pháp kế toán, rằng ta thanh toán trước. Thường sẽ tiện dụng hơn nếu ta định nghĩa $\Phi(D_0)$ là 0 rồi chứng tỏ $\Phi(D_i) \geq 0$ với tất cả *i*. (Bài tập 18.3-1 có nêu một cách dễ dàng để điều quản các trường hợp ở đó $\Phi(D_0) \neq 0$.)

Theo trực giác, nếu hiệu thế $\Phi(D_i) - \Phi(D_{i-1})$ của phép toán thứ i là dương, thì mức hao phí khấu trừ $\hat{c}_i$ biểu thị cho một khoản tính công dư đối với phép toán thứ *i*, và thế của cấu trúc dữ liệu sẽ gia tăng. Nếu hiệu thế là âm, thì mức hao phí khấu trừ biểu thị cho một khoản tính công thiếu đối với phép toán thứ *i*, và mức hao phí thực tế của phép toán được thanh toán bởi sự giảm trong thế.

Các mức hao phí khấu trừ được định nghĩa bởi các phương trình (18.1) và (18.2) sẽ tùy thuộc vào sự chọn lựa của hàm thế (Φ. Các hàm thế khác nhau có thể cho ra các mức hao phí khấu trừ khác nhau mà vẫn là các cận trên trên các mức hao phí thực tế. Thường có sự trả giá cho việc chọn một hàm thế; hàm thế thích hợp nhất thường tùy thuộc vào các cận thời gian mong muốn.

Các phép toán ngăn xếp

Để minh họa phương pháp thế, một lần nữa ta quay lại với ví dụ về các phép toán ngăn xếp PUSH, POP, và MULTIPOP. Ta định nghĩa hàm thế Φ trên một ngăn xếp là số lượng các đối tượng trong ngăn xếp. Với ngăn xếp trống D_0 mà ta bắt đầu, ta có $\Phi(D_0) = 0$. Bởi số lượng các đối tượng trong ngăn xếp không bao giờ là âm, ngăn xếp D_i kết quả sau phép toán thứ *i* có thế không âm, và như vậy

$$\Phi(D_i) \geq 0$$

$$= \Phi(D_0).$$

Do đó, tổng mức hao phí khấu trừ của n phép toán đối với Φ biểu thị cho một cận trên trên mức hao phí thực tế.

Giờ đây ta hãy tính toán các mức hao phí khấu trừ của các phép toán ngăn xếp khác nhau. Nếu phép toán thứ i trên một ngăn xếp chứa s đối tượng là một phép toán PUSH, thì hiệu thế là

$$\begin{aligned}\Phi(D_i) - \Phi(D_{i-1}) &= (s+1) - s \\ &= 1.\end{aligned}$$

Theo phương trình (18.1), mức hao phí khấu trừ của phép toán PUSH này

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + 1 \\ &= 2.\end{aligned}$$

Giả sử rằng phép toán thứ i trên ngăn xếp là MULTIPOP(S, k) và $k' = \min(k, s)$ đối tượng được kéo ra khỏi ngăn xếp. Mức hao phí thực tế của phép toán là k', và hiệu thế là

$$\Phi(D_i) - \Phi(D_{i-1}) = -k'.$$

Như vậy, mức hao phí khấu trừ của phép toán MULTIPOP là

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= k' - k' \\ &= 0.\end{aligned}$$

Cũng vậy, mức hao phí khấu trừ của một phép toán POP bình thường là 0.

Mức hao phí khấu trừ của mỗi trong số ba phép toán là $O(1)$, và như vậy tổng mức hao phí khấu trừ của một dãy n phép toán là $O(n)$. Bởi ta đã chứng tỏ rằng $\Phi(D_i) \geq \Phi(D_0)$, nên tổng mức hao phí khấu trừ của n phép toán là một cận trên trên tổng mức hao phí thực tế. Do đó, mức hao phí ca xấu nhất của n phép toán là $O(n)$.

Gia số một bộ đếm nhị phân

Để có một ví dụ khác về phương pháp thế, ta lại xét tiến trình gia số một bộ đếm nhị phân. Thời gian này, ta định nghĩa thế của bộ đếm sau khi phép toán INCREMENT thứ i là b_i, số lượng 1 trong bộ đếm sau phép toán thứ i.

Ta hãy tính toán mức hao phí khấu trừ của một phép toán INCREMENT. Giả sử phép toán INCREMENT thứ i chỉnh lại t_i bit. Do đó, mức

hao phí thực tế của phép toán tối đa là $t_i + 1$, bởi ngoài việc chỉnh lại t_i bit, nó xác lập tối đa một bit theo một 1. Do đó, số lượng 1 trong bộ đếm sau phép toán thứ i là $b_i \leq b_{i-1} - t_{i+1}$, và hiệu thế là

$$\begin{aligned}\Phi(D_i) - \Phi(D_{i-1}) &\leq (b_{i-1} - t_i + 1) - b_{i-1} \\ &= 1 - t_i .\end{aligned}$$

Do đó, mức hao phí khấu trừ là

$$\begin{aligned}c_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\leq (t_i + 1) + (1 - t_i) \\ &= 2.\end{aligned}$$

Nếu bộ đếm bắt đầu tại zero, thì $\Phi(D_o) = 0$. Bởi $\Phi(D_i) \geq 0$ với tất cả i, tổng mức hao phí khấu trừ của một dãy n phép toán INCREMENT là một cận trên trên tổng mức hao phí thực tế, và như vậy mức hao phí ca xấu nhất của n phép toán INCREMENT là $O(n)$.

Phương pháp thế cho ta một cách dễ dàng để phân tích bộ đếm thậm chí khi nó không bắt đầu tại zero. Thoạt đầu có b_0 1, và sau n phép toán INCREMENT sẽ có b_n 1, ở đó $0 \leq b_0, b_n \leq k$. Ta có thể viết lại phương trình (18.2) dưới dạng

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} (\hat{c}_i - \Phi(D_n) + \Phi(D_0) . \qquad (18.3)$$

Ta có $c_i \leq 2$ với tất cả $1 \leq i \leq n$. Bởi $\Phi(D_0) = b_0$ và $\Phi(D_n) = b_n$, nên tổng mức hao phí thực tế của n phép toán INCREMENT sẽ là

$$\begin{aligned}\sum_{i=1}^{n} c_i &\leq \sum_{i=1}^{n} 2 - b_n + b_0 \\ &= 2n - b_n + \mathrm{b}_0 .\end{aligned}$$

Đặc biệt, lưu ý do $b_0 \leq k$, nếu ta thi hành ít nhất $n = \Omega(k)$ phép toán INCREMENT, tổng mức hao phí thực tế là $O(n)$, bất kể bộ đếm chứa giá trị ban đầu nào.

Bài tập

18.3-1

Giả sử ta có một hàm thế Φ sao cho $\Phi(D_i) \geq \Phi(D_0)$ với tất cả i, nhưng $\Phi(D_0) \neq 0$. Chứng tỏ ở đó tồn tại một hàm thế (Φ' sao cho $\Phi'(D_0) = 0$, $\Phi'(D_i) \geq 0$ với tất cả $i \geq 1$, và các mức hao phí khấu trừ dùng Φ' cũng giống như các mức hao phí khấu trừ dùng 4).

18.3-2

Làm lại Bài tập 18.1-3 dùng một phương pháp thế của phân tích.

18.3-3

Xét một cấu trúc dữ liệu đống nhị phân bình thường với n thành phần hỗ trợ các chỉ lệnh INSERT và EXTRACT-MIN trong $O(\lg n)$ thời gian trường hợp xấu nhất. Nêu một hàm thế Φ sao cho mức hao phí khấu trừ của INSERT là $O(\lg n)$ và mức hao phí khấu trừ của EXTRACT-MIN là $O(1)$, và chứng tỏ nó làm việc.

18.3-4

Đâu là tổng mức hao phí thi hành n phép toán ngăn xếp PUSH, POP, và MULTIPOP, giả định ngăn xếp bắt đầu bằng s_0 đối tượng và hoàn tất với s_n đối tượng?

18.3-5

Giả sử rằng một bộ đếm bắt đầu tại một số với b 1 trong phần biểu thị nhị phân của nó, thay vì tại 0. Chứng tỏ mức hao phí thực hiện n phép toán INCREMENT là $O(n)$ nếu $n = \Omega(b)$. (Đừng mặc nhận b là hằng số.)

18.3-6

Nêu cách thực thi một hàng đợi với hai ngăn xếp bình thường (Bài tập 11.1-6) sao cho mức hao phí khấu trừ của mỗi phép toán ENQUEEE và mỗi phép toán DEQUEUE là $O(1)$.

18.4 Các bảng động

Trong vài ứng dụng, ta không biết trước có bao nhiêu đối tượng sẽ được lưu trữ trong một bảng. Ta có thể phân bổ không gian cho một bảng, để rồi về sau phát hiện nó không đủ. Như vậy, bảng phải được phân bổ lại với một kích cỡ lớn hơn, và tất cả các đối tượng được lưu trữ trong bảng ban đầu phải được chép vào bảng mới, lớn hơn. Cũng vậy, nếu nhiều đối tượng đã được xóa ra khỏi bảng, ta cũng nên phân bổ lại theo một kích cỡ nhỏ hơn. Trong đoạn này, ta nghiên cứu bài toán năng động mở rông và rút gọn một bảng. Dùng phép phân tích khấu trừ, ta sẽ chứng tỏ mức hao phí khấu trừ của phép chèn và phép xóa chỉ là $O(1)$, cho dù mức hao phí thực tế của một phép toán là lớn khi nó ứng tác một phép mở rộng hay phép rút gọn. Hơn nữa, ta sẽ xem làm thế nào để bảo đảm không gian còn rảnh trong một bảng động không bao giờ vượt quá một số bất biến của tổng không gian.

Ta mặc nhận rằng bảng động hỗ trợ các phép toán TABLE-INSERT và TABLE-DELETE. TABLE-INSERT chèn vào bảng một mục choán một ***khe*** đơn lẻ, nghĩa là, một không gian cho một mục. Cũng vậy, ta có thể xem TABLE-DELETE như là gỡ bỏ một mục ra khỏi bảng, đồng thời giải phóng một khe. Các chi tiết của phương pháp cấu trúc dữ liệu được dùng để tổ chức bảng là không quan trọng; ta có thể dùng một ngăn xếp (Đoạn 11.1), một đống (Đoạn 7.1), hoặc một bảng ánh số (Chương 12). Cũng có thể dùng một mảng hoặc tập hợp các mảng để thực thi kho lưu trữ đối tượng, như ta đã làm trong Đoạn 11.3.

Để tiện dụng, ta dùng một khái niệm đã giới thiệu trong phần phân tích kỹ thuật ánh số (Chương 12). Ta định nghĩa ***hệ số tải*** $\alpha(T)$ của một bảng không trống T là số lượng các mục lưu trữ trong bảng chia cho kích cỡ (số lượng các khe) của bảng. Ta gán cho bảng trống (bảng không có mục nào) kích cỡ 0, và ta định nghĩa hệ số tải của nó là 1. Nếu hệ số tải của một bảng động được định cận dưới theo một hằng, thì không gian còn rảnh trong bảng sẽ không bao giờ nhiều hơn một phân số bất biến của tổng lượng không gian.

Để bắt đầu, ta phân tích một bảng động ở đó ta chỉ thực hiện các phép chèn. Sau đó, ta xét trường hợp chung hơn ở đó cả phép chèn lẫn phép xóa đều được phép.

18.4.1 Mở rộng bảng

Ta hãy mặc nhận rằng kho lưu trữ cho một bảng được phân bổ dưới dạng một mảng các khe. Một bảng đầy khi tất cả các khe đã được dùng hoặc, tương đương, khi hệ số tải của nó là[1]. Trong vài môi trường phần mềm, nếu cố chèn một mục vào một bảng đã đầy, ta không có cách nào khác ngoài việc phải bỏ ngang cùng với một lỗi. Tuy nhiên, ta mặc nhận môi trường phần mềm, cũng như nhiều môi trường hiện đại, sẽ cung cấp một hệ quản lý bộ nhớ có thể phân bổ và giải phóng các khối lưu trữ theo yêu cầu. Như vậy, khi một mục được chèn vào một bảng đầy, ta có thể ***mở rộng*** bảng bằng cách phân bổ một bảng mới có nhiều khe hơn bảng cũ rồi chép các mục từ bảng cũ vào bảng mới.

Một phép phỏng đoán chung đó là phân bổ một bảng mới có số khe nhiều gấp hai lần so với bảng cũ. Nếu chỉ thực hiện các phép chèn, hệ số tải của một bảng luôn ít nhất là 1/2, và như vậy lượng không gian lãng phí không bao giờ vượt quá nửa tổng không gian trong bảng.

Trong mã giả dưới đây, ta mặc nhận rằng T là một đối tượng biểu thị

1 Trong vài tình huống, như một bảng ánh số địa chỉ mở, có thể ta phải xem một bảng là đầy nếu hệ số tải của nó bằng một hằng hoàn toàn nhỏ hơn 1. (Xem Bài tập 18.4-2.)

bảng. Trường *table*[*T*] chứa một biến trỏ đến khối lưu trữ biểu thị cho bảng. Trường *num*[*T*] chứa số lượng các mục trong bảng, và trường *size*[*T*] là tổng của các khe trong bảng. Thoạt đầu, bảng là trống: *num*[*T*] = *size*[*T*] = 0.

```
TABLE-INSERT (T, x)
1   if size[ T ] = 0
2        then  phân bổ table[T] bằng 1 khe
3              size[T] ← 1
4   if num[T] = size[ T]
5        then phân bổ new-table bằng 2 . size[ T] các khe
6              chèn tất cả các mục trong table[T] vào new-table
7              giải phóng table[T]
8              table[T] ← new-table
9              size[ T] ← 2 . size[ T]
10  chèn x vào table[T]
11  num[T] ← num[T] + 1
```

Lưu ý, ta có hai thủ tục "chèn" ở đây: chính thủ tục TABLE-INSERT và ***phép chèn cơ bản*** vào một bảng trong các dòng 6 và 10. Ta có thể phân tích thời gian thực hiện của TABLE-INSERT theo dạng số lượng phép chèn cơ bản bằng cách gán một mức hao phí 1 cho từng phép chèn cơ bản. Ta mặc nhận rằng thời gian thực hiện thực tế của TABLE-INSERT là tuyến tính theo thời gian để chèn các mục riêng lẻ, sao cho phần việc chung để phân bổ một bảng ban đầu trong dòng 2 là bất biến và phần việc chung để phân bổ và giải phóng kho lưu trữ trong các dòng 5 và 7 bị chi phối bởi mức hao phí chuyển giao các mục trong dòng 6. Ta gọi sự kiện ở đó mệnh đề **then** trong các dòng 5-9 được thi hành là một ***phép mở rộng*** [expension].

Ta hãy phân tích một dãy *n* phép toán TABLE-INSERT trên một bảng trống từ đầu. Đâu là mức hao phí c_i của phép toán thứ *i*? Nếu có chỗ trong bảng hiện hành (hoặc giả đây là phép toán đầu tiên), thì c_i = 1, bởi ta chỉ cần thực hiện một phép chèn cơ bản trong dòng 10. Tuy nhiên, nếu bảng hiện hành đầy, và một phép mở rộng xảy ra, thì $c_i = i$: mức hao phí là 1 cho phép chèn cơ bản trong dòng 10 cộng với i - 1 cho các mục phải được chép từ bảng cũ sang bảng mới trong dòng 6. Nếu *n* phép toán được thực hiện, mức hao phí ca xấu nhất của một phép toán là $O(n)$, dẫn đến một cận trên $O(n^2)$ trên tổng thời gian thực hiện của *n* phép toán.

Cận này không chặt, bởi mức hao phí của tiến trình mở rộng bảng thường không phát sinh trong quá trình diễn biến của n phép toán TABLE-INSERT. Cụ thể, phép toán thứ i chỉ đưa đến một phép mở rộng khi $i - 1$ là một lũy thừa chính xác của 2. Mức hao phí khấu trừ của một phép toán thực tế là $O(1)$, như ta có thể chứng tỏ bằng phương pháp kết tập. Mức hao phí của phép toán thứ i là

$$c_i = \begin{cases} i \text{ nếu } i - 1 \text{ là một lũy thừa chính xác của 2}, \\ \text{bằng không nó là } 1. \end{cases}$$

Do đó, tổng mức hao phí của n phép toán TABLE-INSERT là

$$\begin{aligned} \sum_{i=1}^{n} c_i &\leq n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &< n + 2n \\ &= 3n , \end{aligned}$$

bởi có tối đa n phép toán có mức hao phí là 1 và các mức hao phí của các phép toán còn lại sẽ hình thành một sêri cấp số nhân. Bởi tổng mức hao phí của n phép toán TABLE-INSERT là $3n$, nên mức hao phí khấu trừ của một phép toán đơn lẻ là 3.

Nếu dùng phương pháp kế toán, ta có thể cảm nhận được tại sao mức hao phí khấu trừ của một phép toán TABLE-INSERT phải là 3. Theo trực giác, mỗi mục thanh toán cho 3 phép chèn cơ bản: chèn chính nó trong bảng hiện hành, dời chính nó khi bảng mở rộng, và dời một mục khác đã được dời một lần khi bảng mở rộng. Ví dụ, giả sử kích cỡ của bảng là m liền sau một phép mở rộng. Như vậy, số lượng các mục trong bảng là $m/2$, và bảng không chứa khoản tín dụng nào. Ta tính công 3 đôla cho mỗi phép chèn. Phép chèn cơ bản xảy ra tức thời có mức hao phí là 1 đôla. Một đôla khác được đặt trên mục đã chèn dưới dạng một khoản tín dụng. Đôla thứ ba được đặt trên một trong $m/2$ mục sẵn có trong bảng dưới dạng một khoản tín dụng. Việc điền đầy bảng yêu cầu $m/2$ phép chèn bổ sung, và như vậy, cho đến lúc đó bảng chứa m mục và đầy, mỗi mục có một đôla để thanh toán cho phép chèn lại của nó trong khi mở rộng.

Phương pháp thế cũng có thể được dùng để phân tích một dãy n phép toán TABLE-INSERT, và ta sẽ dùng nó trong Đoạn 18.4.2 để thiết kế một phép toán TABLE-DELETE cũng có mức hao phí khấu trừ $O(1)$. Để bắt đầu, ta định nghĩa một hàm thế Φ bằng 0 liền sau một phép mở rộng nhưng tạo nên kích cỡ bảng vào lúc bảng đầy, sao cho phép mở rộng kế tiếp có thể được thanh toán bởi thế. Hàm

$$\Phi(T) = 2 \cdot num[T] - size[T] \tag{18.4}$$

là một khả năng. Liền sau một phép mở rộng, ta có $num[T] = size[T]/2$, và như vậy $\Phi(T) = 0$, như mong muốn. Liền trước một phép mở rộng, ta có $num[T] = size[T]$, và như vậy $\Phi(T) = num[T]$, như mong muốn. Giá trị ban đầu của thế là 0, và do bảng luôn ít nhất đầy phân nửa, nên $num[T] \geq size[T]/2$, hàm ý $\Phi(T)$ luôn không âm. Như vậy, tổng các mức hao phí khấu trừ của n phép toán TABLE-INSERT là một cận trên trên tổng các mức hao phí thực tế.

Để phân tích mức hao phí khấu trừ của phép toán TABLE-INSERT thứ i, ta cho num_i thể hiện số lượng các mục được lưu trữ trong bảng sau phép toán thứ i, $size_i$ thể hiện tổng kích cỡ của bảng sau phép toán thứ i, và Φ thể hiện thế [potential] sau phép toán thứ i. Thoạt đầu, ta có $num_0 = 0$, $size_0 = 0$, và $\Phi_0 = 0$.

Nếu phép toán TABLE-INSERT thứ i không ứng tác một phép mở rộng, thì $size_i = size_{i-1}$, và mức hao phí khấu trừ của phép toán là

$$
\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (2 \bullet num_i - size_i) - (2 \bullet num_{i-1} - size_{i-1}) \\
&= 1 + (2 \bullet num_i - size_i) - (2(num_i - 1) - size_i) \\
&= 3.
\end{aligned}
$$

Nếu phép toán thứ i ứng tác một phép mở rộng, thì $size_i/2 = size_{i-1} = num_i - 1$, và mức hao phí khấu trừ của phép toán là

$$
\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= num_i + (2 \bullet num_i - size_i) - (2 \bullet num_{i-1} - size_{i-1}) \\
&= num_i + (2 \bullet num_i - (2 \bullet num_i - 2)) - (2(num_i - 1) - (num_i - 1)) \\
&= num_i + 2 - (num_i - 1) \\
&= 3 .
\end{aligned}
$$

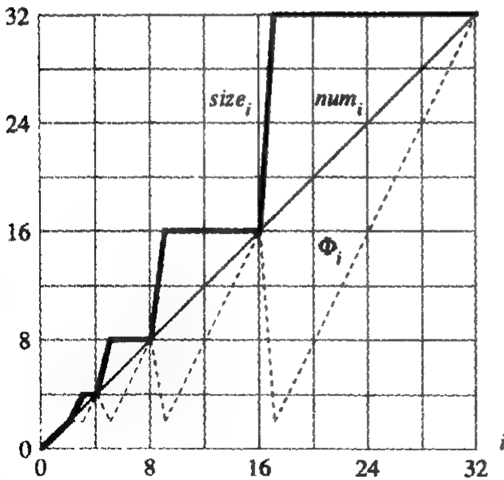
Hình 18.3 chấm vẽ các giá trị của num_i, $size_i$, và Φ_i đối lại i. Lưu ý cách thức mà thế xây dựng để thanh toán cho phép mở rộng của bảng.

18.4.2 Mở rộng và rút gọn bảng

Để thực thi một phép toán TABLE-DELETE, ta chỉ cần gỡ bỏ mục đã chỉ định ra khỏi bảng. Tuy nhiên, thông thường ta nên ***rút gọn*** bảng khi hệ số tải của bảng trở thành quá nhỏ, sao cho không gian lãng phí không quá cao. Phép rút gọn bảng tương tự như phép mở rộng bảng: khi số lượng các mục trong bảng tụt xuống quá thấp, ta phân bổ một bảng mới, nhỏ hơn rồi chép các mục từ bảng cũ vào bảng mới. Sau đó, có thể giải phóng kho lưu trữ cho bảng cũ bằng cách trả nó về hệ quản lý bộ nhớ. Về lý tưởng, ta muốn bảo toàn hai tính chất:

- Hệ số tải của bảng động được định cận dưới theo một hằng, và

- Mức hao phí khấu trừ của một bảng phép toán được định cận trên theo một hằng.



Hình 18.3 Hiệu ứng của một dãy *n* phép toán TABLE-INSERT trên số num_i của các mục trong bảng, số $size_i$ của các khe trong bảng, và thế $\Phi_i = 2 \bullet num_i - size_i$, tất cả đều được tính sau phép toán thứ *i*. Đường kẻ mảnh nêu num_i, đường kẻ dày nêu $size_i$, và đường gạch cách nêu Φ_i. Lưu ý, liền trước một phép mở rộng, thế đã hình thành theo số lượng các mục trong bảng, và do đó nó có thể thanh toán cho việc dời tất cả các mục sang bảng mới. Sau đó, thế tụt xuống đến 0, nhưng lập tức nó được tăng lên 2 khi chèn mục do phép mở rộng gây ra.

Ta mặc nhận mức hao phí có thể được tính theo dạng các phép chèn và phép xóa cơ bản.

Một chiến lược tự nhiên để mở rộng và rút gọn đó là nhân đôi kích cỡ bảng khi một mục được chèn vào một bảng đầy và chia đôi kích cỡ khi một phép xóa khiến bảng chỉ còn đầy ít hơn một nửa. Chiến lược này bảo đảm hệ số tải của bảng không bao giờ tụt xuống dưới 1/2, nhưng đáng tiếc, nó có thể khiến mức hao phí khấu trừ của một phép toán trở nên khá lớn. Xét bối cảnh dưới đây. Ta thực hiện *n* phép toán trên một bảng *T*, ở đó *n* là một lũy thừa chính xác của 2. *n*/2 phép toán đầu tiên là các phép chèn, mà theo phân tích trên đây của chúng ta có tổng mức hao phí là $\Theta(n)$. Ở cuối dãy phép chèn này, $num[T] = size[T] = n/2$. Với *n*/2 phép toán thứ hai, ta thực hiện dãy dưới đây:

I, D, D, I, I, D, D, I, I,...,

ở đó I thay cho một phép chèn và D thay cho một phép xóa. Phép chèn đầu tiên khiến một phép mở rộng của bảng theo kích cỡ *n*. Hai

phép xóa theo sau khiến một phép rút gọn bảng trở về kích cỡ $n/2$. Hai phép chèn sau đó khiến một phép mở rộng khác, và vân vân. Mức hao phí của mỗi phép mở rộng và phép rút gọn là $\Theta(n)$, và có $\Theta(n)$ của chúng. Như vậy, tổng mức hao phí của n phép toán là $\Theta(n^2)$, và mức hao phí khấu trừ của một phép toán là $\Theta(n)$.

Chiến lược này hiển nhiên là khó: sau một phép mở rộng, ta không thực hiện đủ các phép xóa để thanh toán cho một phép rút gọn. Cũng vậy, sau một dạng rút gọn, ta không thực hiện đủ các phép chèn để thanh toán cho một phép mở rộng.

Để cải thiện chiến lược này, ta cho phép hệ số tải của bảng tụt xuống dưới 1/2. Cụ thể, ta tiếp tục nhân đôi kích cỡ bảng khi một mục được chèn vào một bảng đầy, nhưng ta chia đôi kích cỡ bảng khi một phép xóa khiến bảng nhỏ hơn mức đầy 1/4, thay vì mức đầy 1/2 như trước đây. Do đó, hệ số tải của bảng được định cận dưới theo hằng 1/4. Ý tưởng đó là sau một phép mở rộng, hệ số tải của bảng là 1/2. Như vậy, phân nửa các mục trong bảng phải được xóa trước khi một phép rút gọn có thể xảy ra, bởi dạng rút gọn không xảy ra trừ phi hệ số tải tụt xuống dưới 1/4. Cũng vậy, sau một phép rút gọn, hệ số tải của bảng cũng là 1/2. Như vậy, số lượng các mục trong bảng phải được nhân đôi bằng các phép chèn trước khi một phép mở rộng có thể xảy ra, bởi phép mở rộng chỉ xảy ra khi hệ số tải vượt quá 1.

Ta bỏ qua mã của TABLE-DELETE, bởi nó tương tự như TABLE-INSERT. Tuy nhiên, để tiện phân tích, ta mặc nhận nếu số lượng các mục trong bảng tụt xuống 0, kho lưu trữ của bảng được giải phóng. Nghĩa là, nếu $num[T] = 0$, thì $size[T] = 0$.

Giờ đây ta có thể dùng phương pháp thế để phân tích mức hao phí của một dãy n phép toán TABLE-INSERT và TABLE-DELETE. Ta bắt đầu bằng cách định nghĩa một hàm thế Φ là 0 liền sau một phép mở rộng hoặc phép rút gọn và xây dựng khi hệ số tải tăng lên 1 hoặc giảm xuống 1/4. Ta hãy thể hiện hệ số tải của một bảng không trống T bằng $\alpha(T) = num[T]/size[T]$. Bởi với một bảng trống, $num[T] = size[T] = 0$ và $\alpha[T] = 1$, ta luôn có $num[T] = \alpha(T) \cdot size[T]$, dẫu bảng có trống hay không. Ta sẽ dùng làm hàm thế

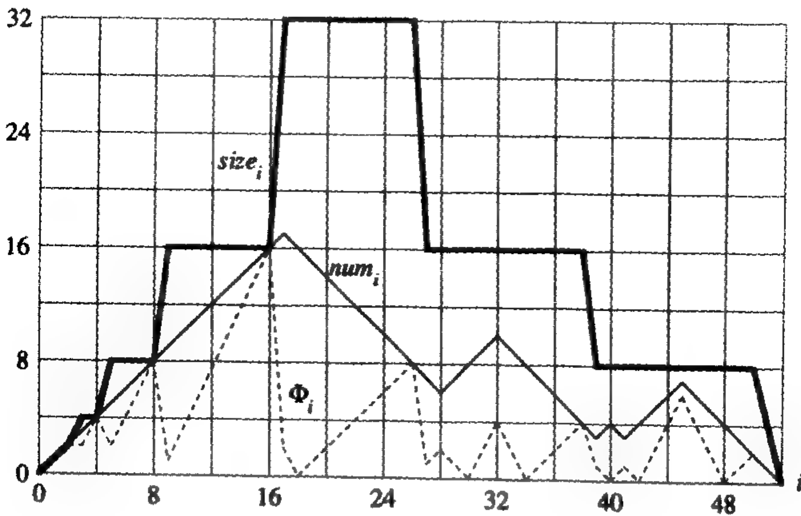
$$\Phi(T) = \begin{cases} 2 \cdot num[T] - size[T] & \text{nếu } \alpha(T) \geq 1/2\,, \\ size[T]/2 - num[T] & \text{nếu } \alpha(T) < 1/2\,. \end{cases} \quad (18.5)$$

Nhận thấy thế của một bảng trống là 0 và thế không bao giờ âm. Như vậy, tổng mức hao phí khấu trừ của một dãy các phép toán đối với Φ là một cận trên trên mức hao phí thực tế của chúng.

Trước khi tiếp tục với một phân tích chính xác, ta tạm ngưng để xem

xét vài tính chất của hàm thế. Lưu ý, khi hệ số tải là 1/2, thế là 0. Khi nó là 1, ta có $size[T] = num[T]$, hàm ý $\Phi(T) = num[T]$, và như vậy thế có thể thanh toán cho một phép mở rộng nếu một mục được chèn. Khi hệ số tải là 1/4, ta có $size[T] = 4 \bullet num[T]$, hàm ý $\Phi(T) = num[T]$, và như vậy thế có thể thanh toán cho một phép rút gọn nếu xóa một mục. Hình 18.4 minh họa cách ứng xử của thế đối với một dãy các phép toán.

Để phân tích một dãy *n* phép toán TABLE-INSERT và TABLE-DELETE, ta cho c_i thể hiện mức hao phí thực tế của phép toán thứ *i*, $\hat{c}_i$ thể hiện mức hao phí khấu trừ của nó đối với Φ, num_i thể hiện số lượng các mục được lưu trữ trong bảng sau phép toán thứ *i*, $size_i$ thể hiện tổng kích cỡ của bảng sau phép toán thứ *i*, α_i thể hiện hệ số tải của bảng sau phép toán thứ *i*, và Φ_i thể hiện thế sau phép toán thứ *i*. Thoạt đầu, $num_0 = 0$, $size_0 = 0$, $\alpha_0 = 1$, và $\Phi_0 = 0$.



Hình 18.4 Hiệu ứng của một dãy *n* phép toán TABLE-INSERT và TABLE-DELETE trên số num_i của các mục trong bảng, số $size_i$ của các khe trong bảng, và thế

$$\Phi_i = \begin{cases} 2 \bullet num_i - size_i & \text{nếu } \alpha_i \geq 1/2 , \\ size_i/2 - num_i & \text{nếu } \alpha_i < 1/2 , \end{cases}$$

tất cả đều được tính sau phép toán thứ *i*. Đường kẻ mảnh nêu num_i, đường kẻ đày nêu $size_i$, và đường gạch cách nêu Φ_i. Lưu ý, liền trước một phép mở rộng, thế đã hình thành theo số lượng các mục trong bảng, và do đó nó có thể thanh toán cho việc dời tất cả các mục sang bảng mới. Cũng vậy, liền trước một dạng rút gọn, thế đã hình thành theo số lượng các mục trong bảng.

Ta bắt đầu với trường hợp ở đó phép toán thứ *i* là TABLE-INSERT.

Nếu $\alpha_{i-1} \geq 1/2$, phép phân tích giống như trường hợp bảng mở rộng trong Đoạn 18.4.1. Dẫu bảng có mở rộng hay không, mức hao phí khấu trừ c_i của phép toán tối đa là 3. Nếu $\alpha_{i-1} < 1/2$, bảng không thể mở rộng làm kết quả của phép toán, bởi phép mở rộng chỉ xảy ra khi $\alpha_{i-1} = 1$. Nếu $\alpha_i < 1/2$, thì mức hao phí khấu trừ của phép toán thứ i là

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\
&= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i - 1)) \\
&= 0 .
\end{aligned}$$

Nếu $\alpha_{i-1} < 1/2$ nhưng $\alpha_i \geq 1/2$, thì

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (2 \bullet num_i - size_i) - (size_{i-1}/2 - num_{i-1}) \\
&= 1 + (2(num_{i-1} + 1) - size_{i-1}) - (size_{i-1}/2 - num_{i-1}) \\
&= 3 \bullet num_{i-1} - \frac{3}{2} size_{i-1} + 3 \\
&= 3\alpha_{i-1} size_{i-1} - \frac{3}{2} size_{i-1} + 3 \\
&< \frac{3}{2} size_{i-1} - \frac{3}{2} size_{i-1} + 3 \\
&= 3.
\end{aligned}$$

Như vậy, mức hao phí khấu trừ của một phép toán TABLE-INSERT tối đa là 3.

Giờ đây ta quay về trường hợp ở đó phép toán thứ i là TABLE-DELETE. Trong trường hợp này, $num_i = num_{i-1} - 1$. Nếu $\alpha_{i-1} < 1/2$, thì ta phải xét phép toán có tạo một phép rút gọn hay không. Nếu không, thì $size_i = size_{i-1}$, và mức hao phí khấu trừ của phép toán là

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\
&= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i + 1)) \\
&= 2.
\end{aligned}$$

Nếu $\alpha_{i-1} < 1/2$ và phép toán thứ i ứng tác một phép rút gọn, thì mức hao phí thực tế của phép toán là $c_i = num_i + 1$, bởi ta xóa một mục và dời num_i mục. Ta có $size_i/2 = size_{i-1}/4 = num_i + 1$, và mức hao phí khấu trừ của phép toán là

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1}$$

$= (num_i + 1) + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1})$

$= (num_i + 1) + ((num_i + 1) - num_i) - ((2 \bullet num_i + 2) - (num_i + 1))$

$= 1.$

Khi phép toán thứ i là một TABLE-DELETE và $\alpha_{i-1} \geq 1/2$, mức hao phí khấu trừ cũng được định cận trên theo một hằng. Phân tích được để lại làm Bài tập 18.4-3.

Tóm lại, bởi mức hao phí khấu trừ của mỗi phép toán được định cận trên theo một hằng, nên thời gian thực tế với bất kỳ dãy n phép toán nào trên một bảng động sẽ là $O(n)$.

Bài tập

18.4-1

Chứng tỏ theo trực giác nếu $\alpha_{i-1} \geq 1/2$ và $\alpha_i \leq 1/2$, thì mức hao phí khấu trừ của một phép toán TABLE-INSERT là 0.

18.4-2

Giả sử ta muốn thực thi một bảng ánh số địa chỉ mở, động. Tại sao ta có thể xem bảng là đầy khi hệ số tải của nó đạt một giá trị α hoàn toàn nhỏ hơn 1? Mô tả ngắn gọn cách thức để phép chèn vào một bảng ánh số địa chỉ mở, động, chạy theo cách mà giá trị dự trù của mức hao phí khấu trừ của mỗi phép chèn là $O(1)$. Tại sao giá trị dự trù của mức hao phí thực tế của mỗi phép chèn không nhất thiết là $O(1)$ với tất cả các phép chèn?

18.4-3

Chứng tỏ nếu phép toán thứ i trên một bảng động là TABLE-DELETE và $\alpha_{i-1} \geq 1/2$, thì mức hao phí khấu trừ của phép toán đối với hàm thế (18.5) được định cận trên theo một hằng.

18.4-4

Giả sử thay vì rút gọn một bảng bằng cách chia đôi kích cỡ của nó khi hệ số tải của nó tụt xuống dưới 1/4, ta rút gọnnó bằng cách nhân kích cỡ của nó cho 2/3 khi hệ số tải của nó tụt xuống dưới 1/3. Dùng hàm thế

$$\Phi(T) = |2 \bullet num[T] - size[T]| ,$$

chứng tỏ mức hao phí khấu trừ của một TABLE-DELETE sử dụng chiến lược này được định cận trên theo một hằng.

Các Bài Toán

18-1 Bộ đếm nhị phân đảo bit

Chương 32 xét một thuật toán quan trọng có tên Fast Fourier Transform, hoặc FFT. Bước đầu tiên của thuật toán FFT thực hiện một ***phép hoán vị đảo bit*** trên một mảng nhập liệu $A[0..n-1]$ có chiều dài là $n = 2^k$ với một số nguyên không âm k. Phép hoán vị này tráo các thành phần mà các chỉ số của chúng có các phần biểu thị nhị phân là nghịch đảo với nhau.

Ta có thể diễn tả từng chỉ số a dưới dạng một dãy k-bit $<a_{k-1}, a_{k-2},..., a_0>$, ở đó $a = \sum_{i=0}^{k-1} a_i 2^i$. Ta định nghĩa

$\text{rev}_k(<a_{k-1}, a_{k-2},...,a_0>) = <a_0, a_1,...,a_{k-1})$;

như vậy,

$$\text{rev}_k(a) = \sum_{i=0}^{k-1} a_{k-i-1} 2^i .$$

Ví dụ, nếu $n = 16$ (hoặc, tương đương, $k = 4$), thì $\text{rev}_k(3) = 12$, bởi phần biểu diễn 4-bit của 3 là 0011, mà khi được đảo ngược sẽ cho 1100, phần biểu diễn 4-bit của 12.

a. Cho một hàm rev_k chạy trong $\Theta(k)$ thời gian, viết một thuật toán để thực hiện phép hoán vị đảo bit trên một mảng có chiều dài $n = 2^k$ trong $O(nk)$ thời gian.

Ta có thể dùng một thuật toán dựa trên dạng phân tích khấu trừ để cải thiện thời gian thực hiện của phép hoán vị đảo bit. Ta duy trì một "bộ đếm đảo bit" và một thủ tục BIT-REVERSED-INCREMENT mà, khi cho một giá trị bộ đếm đảo bit a, sẽ tạo ra $rev_k(rev_k(a) + 1)$. Ví dụ, nếu $k = 4$, và bộ đếm đảo bit bắt đầu tại 0, thì các lệnh gọi sau đó đến BIT-REVERSED-INCREMENT sẽ tạo ra dãy

0000, 1000,0100, 1100, 0010, 1010,...= 0, 8, 4, 12, 2, 10,....

b. Giả sử các từ trong máy tính lưu trữ các giá trị k-bit và trong thời gian đơn vị, máy tính của bạn có thể điều tác các giá trị nhị phân bằng các phép toán như chuyển trái hay phải theo các lượng tùy ý, AND cấp bit, OR cấp bit, vân vân. Mô tả một cách thực thi của thủ tục BIT-REVERSED-INCREMENT cho phép thực hiện phép hoán vị đảo bit trên một mảng n thành phần trong một tổng $O(n)$ thời gian.

c. Giả sử bạn có thể chuyển một từ sang trái hay phải theo chỉ một bit trong đơn vị thời gian. Ta vẫn có thể thực thi một phép hoán vị đảo bit $O(n)$-thời gian hay không?

18-2 Khiến đợt tìm nhị phân trở thành động

Đợt tìm nhị phân của một mảng đã sắp xếp sẽ chiếm thời gian tìm lôga, nhưng thời gian để chèn một thành phần mới là tuyến tính theo kích cỡ của mảng. Ta có thể cải thiện thời gian của phép chèn bằng cách duy trì vài mảng đã sắp xếp.

Cụ thể, giả sử ta muốn hỗ trợ SEARCH và INSERT trên một tập hợp n thành phần. Cho $k = \lceil \lg(n+1) \rceil$, và cho phần biểu thị nhị phân của n là $<n_{k-1}, n_{k-2}, ..., n_0>$. Ta có k mảng đã sắp xếp $A_0, A_1, ..., A_{k-1}$, ở đó với $i = 0, 1, ..., k-1$, chiều dài của mảng A_i là 2^i. Mỗi mảng là đầy hoặc trống, tùy thuộc vào $n_i = 1$ hay $n_i = 0$, theo thứ tự nêu trên. Do đó, tổng số lượng các thành phần được lưu giữ trong tất cả k mảng là $\sum_{i=0}^{k-1} n_i 2^i = n$.

Mặc dù mỗi mảng riêng lẻ được sắp xếp, song không có mối quan hệ đặc biệt giữa các thành phần trong các mảng khác nhau.

a. Mô tả cách thực hiện phép toán SEARCH cho cấu trúc dữ liệu này. Phân tích thời gian thực hiện trường hợp xấu nhất của nó.

b. Mô tả cách chèn một thành phần mới vào cấu trúc dữ liệu này. Phân tích các thời gian thực hiện khấu trừ và trường hợp xấu nhất của nó.

c. Mô tả cách thực thi DELETE.

18-3 Các cây cân đối trọng số khấu trừ

Xét một cây tìm nhị phân bình thường được tăng cường bằng cách cộng vào mỗi nút x trường $size[x]$ cho ra số lượng khóa lưu trữ trong cây con có gốc tại x. Cho α là một hằng trong miền giá trị $1/2 \leq \alpha < 1$. Ta nói rằng một nút x đã cho là ***cân đối α*** nếu

$size[left[x]] \leq \alpha \bullet size[x]$

và

$size[right[x]] \leq \alpha \bullet size[x]$.

Cây dưới dạng một toàn thể là *cân đối* α [α-balanced] nếu mọi nút trong cây cân đối α. G. Varghese đã đề xuất cách tiếp cận khấu trừ dưới đây để duy trì các cây có trọng số cân đối.

a. Một cây cân đối 1/2 là, theo một nghĩa nào đó, cân đối theo mức

nó có thể. Cho một nút x trong một cây tìm nhị phân tùy ý, nêu cách tái thiết cây con có gốc tại x sao cho nó trở thành cân đối 1/2. Thuật toán của bạn phải chạy trong thời gian $\Theta(size[x])$, và nó có thể dùng $O(size[x])$ kho lưu trữ phụ.

b. Chứng tỏ việc thực hiện một đợt tìm kiếm trong một cây tìm nhị phân cân đối α n-nút sẽ mất $O(\lg n)$ thời gian trường hợp xấu nhất.

Với phần còn lại của bài toán này, ta mặc nhận hằng α tuyệt đối lớn hơn 1/2. Giả sử INSERT và DELETE được thực thi như bình thường cho một cây tìm nhị phân n-nút, ngoại trừ sau mọi phép toán như vậy, nếu một nut bất kỳ trong cây không còn cân đối α, thì cây con có gốc tại nút cao nhất như vậy trong cây được "tái thiết" sao cho nó trở thành cân đối 1/2.

Ta sẽ phân tích lược đồ tái thiết này bằng phương pháp thế. Với một nút x trong một cây tìm nhị phân T, ta định nghĩa

$$\Delta(x) = |size[left[x]] - size[right[x]]|,$$

và ta định nghĩa thế của T là

$$\Phi(T) = c \sum_{x \in T : \Delta(x) \geq 2} \Delta(x),$$

ở đó c là một hằng đủ lớn tùy thuộc vào a.

c. Chứng tỏ một cây tìm nhị phân bất kỳ có thế không âm và một cây cân đối 1/2 có thế 0.

d. Giả sử rằng m đơn vị của thế có thể thanh toán cho việc tái thiết một cây con m-nút. c phải lớn bao nhiêu theo dạng α để nó có thể mất $O(1)$ thời gian khấu trừ để tái thiết một cây con không cân đối α?

e. Chứng tỏ tiến trình chèn một nút hoặc xóa một nút ra khỏi một cây cân đối α n-nút có mức hao phí là $O(\lg n)$ thời gian khấu trừ.

Ghi chú Chương

Phương pháp kết tập của phân tích khấu trừ đã được dùng bởi Aho, Hopcroft, và Ullman [4]. Tarjan [189] nghiên cứu các phương pháp kế toán và thế của phân tích khấu trừ và trình bày vài ứng dụng. Ông quy phương pháp kế toán cho vài tác giả, bao gồm M. R. Brown, R. E. Tarjan,

S. Huddleston, và K. Mehlhorn. Ông quy phương pháp thế cho D. D. Sleator. Thuật ngữ "khấu trừ" [amortized] là của D. D. Sleator và R. E. Tarjan.

V *Các Cấu Trúc Dữ Liệu Cao Cấp*

Mở đầu

Phần này quay lại việc xem xét các cấu trúc dữ liệu hỗ trợ các phép toán trên các tập hợp động nhưng ở một cấp cao hơn so với Phần III. Ví dụ, hai trong số các chương vận dụng rộng rãi các kỹ thuật phân tích khấu trừ mà ta đã gặp trong Chương 18.

Chương 19 trình bày các cây B, là các cây tìm kiếm cân đối được thiết kế để lưu trữ trên các đĩa từ. Bởi các đĩa từ vận hành chậm hơn nhiều so với bộ nhớ truy cập ngẫu nhiên, ta đo khả năng thực hiện của các cây B không những bằng quãng thời gian tính toán mà các phép toán tập hợp động tiêu thụ mà còn bằng số lần truy cập đĩa được thực hiện. Với mỗi phép toán cây B, số lần truy cập đĩa gia tăng theo chiều cao của cây B, được các phép toán cây B duy trì ở mức thấp.

Các chương 20 và 21 đề cập các cách thực thi các đống khả trộn [mergeable heaps], hỗ trợ các phép toán INSERT, MINIMUM, EXTRACT-MIN, và UNION. Phép toán UNION hợp nhất, hoặc trộn, hai đống. Các cấu trúc dữ liệu trong các chương này cũng hỗ trợ các phép toán DELETE và DECREASE-KEY.

Các đống nhị thức, xuất hiện trong Chương 20, hỗ trợ từng phép toán này trong $O(\lg n)$ thời gian trường hợp xấu nhất, ở đó n là tổng các thành phần trong đống nhập liệu (hoặc trong hai đống nhập liệu với nhau trong trường hợp UNION). Khi phép toán UNION phải được hỗ trợ thì chính là lúc các đống nhị thức tỏ ra vượt trội so với các đống nhị phân đã giới thiệu trong Chương 7, bởi nó mất $\Theta(n)$ thời gian để hợp nhất hai đống nhị phân trong trường hợp xấu nhất.

Các đống Fibonacci, trong Chương 21, vượt trội hơn các đống nhị thức, ít nhất theo nghĩa lý thuyết. Ta dùng các cận thời gian khấu trừ để đo khả năng thực hiện của các đống Fibonacci. Các phép toán INSERT, MINIMUM, và UNION chỉ mất $O(1)$ thời gian thực tế và khấu trừ trên các đống Fibonacci, và các phép toán EXTRACT-MIN và DELETE mất $O(\lg n)$ thời gian khấu trừ. Tuy nhiên, ưu điểm quan trọng nhất của các đống Fibonacci đó là DECREASE-KEY chỉ mất $O(1)$ thời gian khấu trừ. Thời gian khấu trừ thấp của phép toán DECREASE-KEY chính là lý do tại sao các đống Fibonacci lại là trọng tâm của vài các thuật toán nhanh

nhất theo tiệm cận cho đến giờ cho các bài toán đồ thị.

Cuối cùng, Chương 22 trình bày các cấu trúc dữ liệu cho các tập hợp rời. Ta có một vũ trụ n thành phần được gom nhóm thành các tập hợp động. Thoạt đầu, mỗi thành phần thuộc về tập hợp độc nhất riêng của nó. Phép toán UNION hợp nhất hai tập hợp, và phép truy vấn FIND-SET định danh tập hợp chứa một thành phần đã cho vào lúc đó. Nhờ biểu thị từng tập hợp bằng một cây có gốc đơn giản, ta có được các phép toán nhanh đáng kể: một dãy m phép toán chạy trong $O(m\,\alpha(m, n))$ thời gian, ở đó $\alpha(m, n)$ là một hàm tăng trưởng chậm đáng kinh ngạc—miễn là n vẫn còn là số lượng dự trù của các nguyên tử trong nguyên cả vũ trụ đã biết, $\alpha(m, n)$ tối đa là 4. Tiến trình phân tích khấu trừ chứng minh cận thời gian này cũng phức tạp như cấu trúc dữ liệu là đơn giản. Chương 22 chứng minh một cận đáng quan tâm song có phần đơn giản hơn trên thời gian thực hiện.

•Các chủ đề đề cập trong phần này không chỉ đơn thuần là các ví dụ về các cấu trúc dữ liệu "cao cấp." Các cấu trúc dữ liệu cao cấp khác cũng bao gồm:

•Một cấu trúc dữ liệu do van Emde Boas [194] sáng chế hỗ trợ các phép toán MINIMUM, MAXIMUM, INSERT, DELETE, SEARCH, EXTRACT-MIN, EXTRACT-MAX, PREDECESSOR, và SUCCESSOR trong thời gian trường hợp xấu nhất $O(\lg\lg n)$, chịu sự hạn chế quy định vũ trụ của các khóa là tập hợp $\{1,2,...,n\}$.

•***Các cây động***, đã được Sleator và Tarjan [177] giới thiệu và được Tarjan [188] bàn luận, duy trì một rừng các cây có gốc rời nhau. Mỗi cạnh trong mỗi cây đều có một mức hao phí có giá trị thực. Các cây động hỗ trợ các đợt truy vấn để tìm ra các cha, các gốc, các mức hao phí cạnh, và mức hao phí cạnh cực tiểu trên một lộ trình từ một nút lên đến một gốc. Các cây có thể được điều tác bằng cách cắt các cạnh, cập nhật tất cả các mức hao phí cạnh trên một lộ trình từ một nút lên đến một gốc, nối kết một gốc vào một cây khác, và chuyển một nút thành gốc của cây mà nó xuất hiện trong đó. Một thực thi các cây động cho ra một cận $O(\lg n)$ thời gian khấu trừ cho mỗi phép toán; một thực thi phức tạp hơn cho ra các cận $O(\lg n)$ thời gian trường hợp xấu nhất.

•***Các cây splay***, do Sleator và Tarjan [178] phát triển và Tarjan [188] bàn luận, là một dạng cây tìm nhị phân ở đó các phép toán cây tìm chuẩn chạy trong $O(\lg n)$ thời gian khấu trừ. Một ứng dụng của các cây splay đơn giản hóa các cây động.

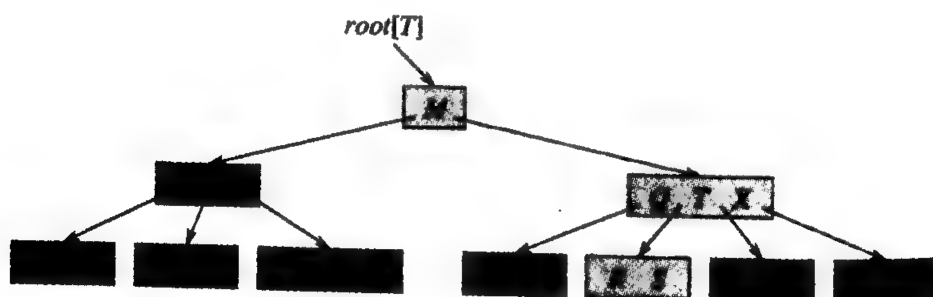
•Các cấu trúc dữ liệu ***ổn định*** [persistent] cho phép các đợt truy vấn, và đôi lúc cả các đợt cập nhật, trên các phiên bản quá khứ của một cấu trúc dữ liệu. Driscoll, Sarnak, Sleator, và Tarjan [59] trình bày các kỹ thuật để duy trì ổn định các cấu trúc dữ liệu đã nối kết, với chỉ một mức hao phí thời gian và không gian nhỏ. Bài toán 14-1 cung cấp một ví dụ đơn giản về một tập hợp động bền.

19 Các Cây B

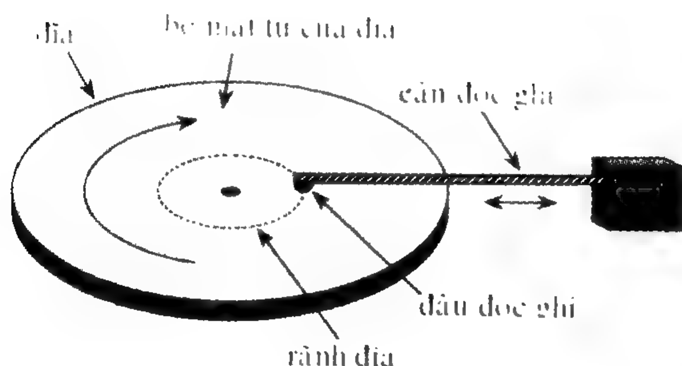
Các cây B là các cây tìm kiếm cân đối được thiết kế để làm việc trên các đĩa từ hoặc các thiết bị lưu trữ thứ cấp truy cập trực tiếp khác. Các cây B tương tự như các cây đỏ đen (Chương 14), nhưng chúng tỏ ra tốt hơn trong việc giảm thiểu các phép toán I/O đĩa.

Các cây B khác với các cây đỏ đen ở chỗ các nút cây B có thể có nhiều con, từ một ít đến hàng ngàn. Nghĩa là, "thừa số rẽ nhánh" của một cây B có thể khá lớn, mặc dù nó thường được xác định bởi các đặc tính của đơn vị đĩa được dùng. Các cây B tương tự như các cây đỏ đen ở chỗ mọi cây B n-nút đều có chiều cao $O(\lg n)$, mặc dù chiều cao của một cây B có thể nhỏ hơn đáng kể so với một cây đỏ đen bởi thừa số rẽ nhánh của nó có thể lớn hơn nhiều. Do đó, các cây B cũng có thể được dùng để thực thi nhiều phép toán tập hợp động trong thời gian $O(\lg n)$.

Các cây B tổng quát hóa các cây tìm nhị phân theo cách tự nhiên. Hình 19.1 nêu một cây B đơn giản. Nếu một nút cây B x chứa $n[x]$ khóa, thì x có $n[x] + 1$ con. Các khóa trong nút x được dùng làm các điểm chia tách biệt miền các khóa được x điều quản thành $n[x] + 1$ miền con, mỗi miền con được điều quản bởi một con của x. Khi tìm kiếm một khóa trong một cây B, ta thực hiện một phép chia $(n[x]+ 1)$ cách dựa trên các phép so sánh với $n[x]$ khóa lưu trữ tại nút x.



Hình 19.1 Một cây B có các khóa là các phụ âm trong tiếng Anh. Một nút trong x chứa $n[x]$ khóa có $n[x] + 1$ con. Tất cả các lá có cùng chiều sâu trong cây. Các nút tô bóng sáng được xem xét trong một đợt tìm kiếm mẫu tự R.



Hình 19.2 Ổ đĩa điển hình.

Đoạn 19.1 cho ta một định nghĩa chính xác về các cây B và chứng minh chiều cao của một cây B chỉ tăng trưởng theo loga với số lượng các nút mà nó chứa. Đoạn 19.2 mô tả cách tìm kiếm một khóa và chèn một khóa vào một cây B, và Đoạn 19.3 mô tả phép xóa. Tuy nhiên, trước khi tiếp tục, ta cần hỏi tại sao các cấu trúc dữ liệu được thiết kế để làm việc trên một đĩa từ lại được đánh giá khác với các cấu trúc dữ liệu được thiết kế để làm việc trong bộ nhớ truy cập ngẫu nhiên chính.

Các cấu trúc dữ liệu trên kho lưu trữ thứ cấp

Có nhiều công nghệ khác nhau sẵn có để cung cấp dung lượng bộ nhớ trong một hệ máy tính. ***Bộ nhớ chính*** của một hệ máy tính thường bao gồm các chip bộ nhớ silicon, mỗi chip có thể lưu giữ 1 triệu bit dữ liệu. Với mỗi bit lưu trữ, công nghệ này đắt tiền hơn công nghệ kho lưu trữ từ tính, như băng từ hoặc đĩa. Một hệ máy tính điển hình có ***kho lưu trữ thứ cấp*** dựa trên các đĩa từ; lượng kho lưu trữ thứ cấp như vậy thường vượt quá lượng bộ nhớ chính theo vài độ lớn.

Hình 19.2 có nêu một ổ đĩa điển hình. Bề mặt đĩa được tráng bằng một chất từ hóa được. Đầu đọc/ghi có thể đọc hoặc ghi dữ liệu theo từ tính trên bề mặt đĩa xoay. Tay đọc/ghi có thể định vị đầu tại các khoảng cách khác nhau từ tâm của đĩa. Khi đầu đứng yên, bề mặt di chuyển bên dưới nó được gọi là một ***rãnh*** [track]. Thông tin được lưu trữ trên mỗi rãnh thường được chia thành một số lương ***trang*** cố định có kích cỡ bằng nhau; với một đĩa điển hình, một trang có thể dài 2048 byte. Đơn vị cơ bản để lưu trữ và truy lục thông tin thường là một trang thông tin—nghĩa là, các lần đọc và ghi đĩa thường là các trang nguyên. ***Thời gian truy cập***—thời gian cần thiết để định vị đầu đọc/ghi và đợi một trang thông tin đã cho di chuyển bên dưới đầu—có thể lớn (ví dụ, 20 miligiây), trong khi thời gian để đọc hoặc ghi một trang, một khi đã truy cập, là nhỏ. Như vậy, giá thanh toán cho mức hao phí thấp về các kỹ thuật lưu

trữ từ tính là thời gian tương đối dài mà nó trải qua để truy cập dữ liệu. Bởi việc di chuyển các âm điện tử dễ hơn nhiều so với việc di chuyển các đối tượng lớn (hoặc thậm chí nhỏ), các thiết bị lưu trữ là hoàn toàn điện tử, như các chip bộ nhớ silicon, có một thời gian truy cập nhỏ hơn nhiều so với các thiết bị lưu trữ có các phần di động, như đĩa từ các ổ đĩa. Tuy nhiên, một khi mọi thứ được định vị đúng đắn, việc đọc hoặc ghi một đĩa từ là hoàn toàn điện tử (ngoài việc quay đĩa), và có thể nhanh chóng đọc hoặc ghi các lượng dữ liệu lớn.

Thông thường, máy tính sẽ mất nhiều thời gian hơn để truy cập một trang thông tin và đọc nó từ một đĩa hơn là xem xét tất cả thông tin được đọc. Vì lý do đó, chương này sẽ xem xét riêng biệt hai thành phần chính của thời gian thực hiện:

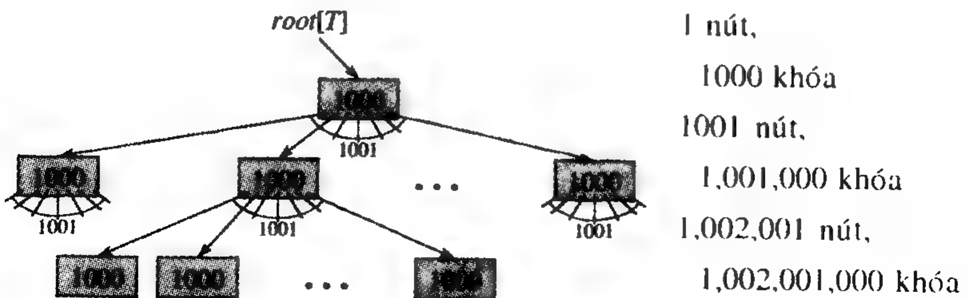
- • số lần truy cập đĩa, và
- • thời gian (tính toán) CPU.

Số lần truy cập đĩa được đo theo dạng số lượng trang thông tin cần có để đọc hoặc ghi vào/ra đĩa. Ta lưu ý thời gian truy cập đĩa không phải là bất biến—nó tùy thuộc vào khoảng cách giữa rãnh hiện hành và rãnh mong muốn và cũng tùy thuộc vào trạng thái quay ban đầu của đĩa. Tuy nhiên, ta sẽ dùng số lượng trang được đọc hoặc được ghi dưới dạng một phép xấp xỉ cấp đầu tiên [first-order] thô của tổng thời gian bỏ ra để truy cập đĩa.

Trong một ứng dụng cây B điển hình, lượng dữ liệu được điều quản thường lớn đến nỗi tất cả dữ liệu không thể vừa trong bộ nhớ chính cùng một lúc. Các thuật toán cây B chép các trang đã lựa từ đĩa vào bộ nhớ chính khi cần và viết trở lại lên trên các trang đĩa đã thay đổi. Bởi các thuật toán cây B chỉ cần một số lượng bất biến các trang trong bộ nhớ chính vào một thời điểm bất kỳ, nên kích cỡ của bộ nhớ chính không giới hạn kích cỡ của các cây B có thể được điều quản.

Ta lập mô hình các phép toán đĩa trong mã giả như sau. Cho x là một biến trỏ đến một đối tượng. Nếu đối tượng hiện nằm trong bộ nhớ chính của máy tính, thì ta có thể tham chiếu các trường của đối tượng như bình thường: *key*[x], chẳng hạn. Tuy nhiên, nếu đối tượng mà x tham chiếu thường trú trên đĩa, ta phải thực hiện phép toán DISK-READ(x) để đọc đối tượng x vào bộ nhớ chính trước khi các trường của nó có thể được tham chiếu. (Ta mặc nhận rằng nếu x đã có sẵn trong bộ nhớ chính, thì DISK-READ(x) không yêu cầu truy cập đĩa; nó là một "không vận hành") Cũng vậy, phép toán DISK-WRITE(x) được dùng để lưu mọi thay đổi đã được thực hiện đối với các trường của đối tượng x. Nghĩa là, khuôn mẫu điển hình để làm việc với một đối tượng là như sau.

```
...
x ← một biến trỏ đến một đối tượng
các phép toán (x)
DISK-READ(x) truy cập và/hoặc sửa đổi các trường của x
DISK-WRITE(x)              ▷ Được bỏ qua nếu không có trường của
                                                    x thay đổi.
khác truy cập nhưng không sửa đổi các trường của x
...
```



Hình 19.3 Một cây B có chiều cao 2 chứa trên một tỷ khóa. Mỗi nút trong và lá chứa 1000 khóa. Có 1001 nút tại độ sâu 1 và trên một triệu lá tại độ sâu 2. Trong mỗi nút *x* là *n*[*x*], số lượng khóa trong *x*.

Hệ thống chỉ có thể lưu giữ một số trang hạn chế trong bộ nhớ chính tại một thời điểm bất kỳ. Ta sẽ mặc nhận các trang không còn dùng sẽ được hệ thống xả sạch ra khỏi bộ nhớ chính; các thuật toán cây B của chúng ta sẽ bỏ qua vấn đề này.

Do trong hầu hết các hệ thống, thời gian thực hiện của một thuật toán cây B chủ yếu được xác định bởi số lượng phép toán DISK-READ và DISK-WRITE mà nó thực hiện, nên quả hợp lý khi ta sử dụng nhiều các phép toán này bằng cách để chúng đọc hoặc ghi càng nhiều thông tin càng tốt. Như vậy, một nút cây B thường lớn bằng cả một trang đĩa. Do đó, số lượng con mà một nút cây B có thể có được hạn chế bởi kích cỡ của một trang đĩa.

Với một cây B lớn được lưu trữ trên một đĩa, các thừa số rẽ nhánh giữa 50 và 2000 thường được dùng, tùy theo kích cỡ của một khóa tương đối với kích cỡ của một trang. Một thừa số rẽ nhánh lớn rút gọn đáng kể cả chiều cao của cây lẫn số lần truy cập đĩa cần thiết để tìm một khóa bất kỳ. Hình 19.3 nêu một cây B có một thừa số rẽ nhánh 1001 và chiều cao 2 có thể lưu trữ trên một tỷ khóa; tuy vậy, do nút gốc có thể được lưu giữ thường trực trong bộ nhớ chính, nên chỉ cần tối đa

hai lần truy cập đĩa để tìm một khóa bất kỳ trong cây này!

19.1 Định nghĩa cây B

Để đơn giản hóa, ta mặc nhận, như với trường hợp các cây tìm nhị phân và các cây đỏ đen, rằng mọi "thông tin vệ tinh" kết hợp với một khóa được lưu trữ trong cùng nút với khóa. Trong thực tế, ta có thể thực tế lưu trữ với mỗi khóa chỉ một biến trỏ đến một trang đĩa khác chứa thông tin vệ tinh cho khóa đó. Mã giả trong chương này mặc nhận thông tin vệ tinh kết hợp với một khóa, hoặc biến trỏ đến thông tin vệ tinh đó, di chuyển với khóa mỗi khi khóa được dời từ nút này sang nút khác. Một tổ chức cây B phổ dụng lưu trữ toàn bộ thông tin vệ tinh trong các lá và chỉ lưu trữ các khóa và các biến trỏ con trong các nút trong, như vậy tối đa hóa thừa số rẽ nhánh của các nút trong.

Một ***cây B*** *T* là một cây có gốc (có gốc *root*[*T*]) có các tính chất sau.

1. Mọi nút *x* có các trường dưới đây:

a. $n[x]$, số lượng các khóa hiện được lưu trữ trong nút *x*,

b. chính $n[x]$ khóa, được lưu trữ theo thứ tự không giảm: $key_1[x] \leq key_2[x] \leq \ldots \leq key_{n[x]}[x]$, và

c. $leaf[x]$, một giá trị bool là TRUE nếu *x* là một lá và FALSE nếu *x* là một nút trong.

2. Nếu *x* là một nút trong, nó cũng chứa $n[x] + 1$ biến trỏ $c_1[x]$, $c_2[x], \ldots, c_{n[x]+1}[x]$ đến các con của nó. Các nút lá không có các con, do đó các trường c_i của chúng là không xác định.

3. Các khóa $key_i[x]$ tách biệt các miền khóa được lưu trữ trong mỗi cây con: nếu k_i là một khóa bất kỳ được lưu trữ trong cây con có gốc $c_i[x]$, thì

$$k_1 \leq key_1[x] \leq k_2 \leq key_2[x] \leq \ldots \leq key_{n[x]}[x] \leq k_{n[x]+1}.$$

4. Mọi lá có cùng độ sâu, là chiều cao *h* của cây.

5. Có các cận trên và dưới trên số lượng khóa mà một nút có thể chứa. Có thể diễn tả các cận này theo dạng một số nguyên cố định $t \geq 2$ có tên ***độ cực tiểu*** của cây B:

a. Mọi nút khác ngoài gốc phải có ít nhất $t - 1$ khóa. Như vậy, ngoài gốc mọi nút trong sẽ có ít nhất *t* con. Nếu cây không trống, gốc phải có ít nhất một khóa.

b. Mọi nút có thể chứa tối đa $2t - 1$ khóa. Do đó, một nút trong có thể có tối đa $2t$ con. Ta nói một nút là ***đầy*** nếu nó chứa chính xác $2t - 1$ khóa.

Cây-B đơn giản nhất xảy ra khi $t = 2$. Như vậy, mọi nút trong có 2, 3, hoặc 4 con, và ta có một ***cây 2-3-4***. Tuy nhiên, trong thực tế, ta thường dùng các giá trị lớn hơn nhiều của t.

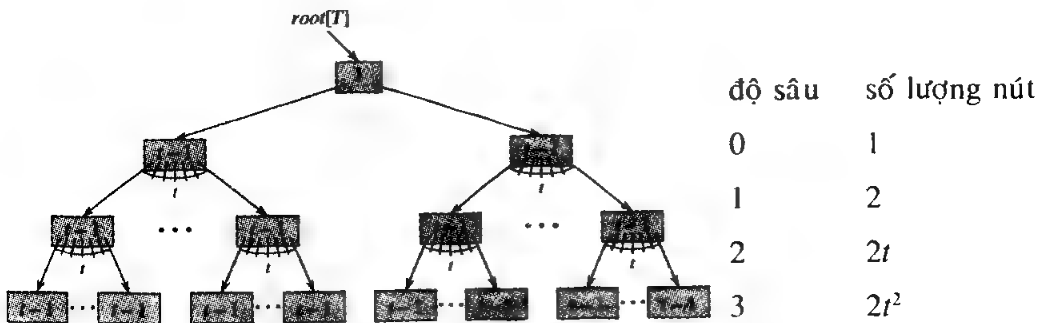
Chiều cao của một cây B

Số lần truy cập đĩa cần thiết cho hầu hết các phép toán trên một cây B thường tỷ lệ với chiều cao của cây B. Giờ đây, ta phân tích chiều cao trường hợp xấu nhất của một cây B.

Định lý 19.1

Nếu $n \geq 1$, thì với bất kỳ cây B T khóa n có chiều cao h và độ cực tiểu $t \geq 2$,

$$h \leq \log_t \frac{n+1}{2}.$$



Hình 19.4 Một cây B có chiều cao 3 chứa một số lượng khóa khả dĩ cực tiểu. Trong mỗi nút x là $n[x]$.

Chứng minh Nếu một cây B có chiều cao h, số lượng nút của nó được giảm thiểu khi gốc chứa một khóa và tất cả các nút khác chứa t - 1 khóa. Trong trường hợp này, có 2 nút ở độ sâu 1, $2t$ nút ở độ sâu 2, $2t^2$ nút ở độ sâu 3, và vân vân, cho đến khi tại độ sâu h ta có $2t^{h-1}$ nút. Hình 19.4 minh họa một cây như vậy với $h = 3$. Như vậy, số n các khóa thỏa bất đẳng thức

$$\begin{aligned} n &\geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} \\ &= 1 + 2(t-1)\left(\frac{t^h - 1}{t - 1}\right) \\ &= 2t^h - 1, \end{aligned}$$

hàm ý định lý.

Ở đây ta thấy năng lực của các cây B, so với các cây đỏ đen. Mặc dù

chiều cao của cây tăng trưởng dưới dạng $O(\lg n)$ trong cả hai trường hợp (chắc bạn còn nhớ t là một hằng), song với các cây B cơ số của lôga có thể lớn hơn gấp nhiều lần. Như vậy, các cây B tiết kiệm một thừa số khoảng $\lg t$ so với các cây đỏ đen theo số lượng nút được xem xét cho hầu hết các phép toán cây. Do việc xem xét một nút tùy ý trong một cây thường yêu cầu một lần truy cập đĩa, nên số lần truy cập đĩa sẽ giảm đáng kể.

Bài tập

19.1-1

Tại sao ta không cho phép một độ cực tiểu $t = 1$?

19.1-2

Với các giá trị nào của t cây trong Hình 19.1 sẽ là một cây B hợp pháp?

19.1-3

Nêu tất cả các cây B hợp pháp có độ cực tiểu 2 biểu diễn {1, 2, 3, 4, 5}.

19.1-4

Suy ra một cận trên chặt trên số lượng các khóa có thể được lưu trữ trong một cây B có chiều cao h dưới dạng một hàm của độ cực tiểu t.

19.1-5

Mô tả cấu trúc dữ liệu kết quả nếu mỗi nút đen trong một cây đỏ đen đã thu hút các con đỏ của nó, sát nhập các con với chính nó.

19.2 Các phép toán cơ bản trên các cây B

Trong đoạn này, ta trình bày các chi tiết của các phép toán B-TREE-SEARCH, B-TREE-CREATE, và B-TREE-INSERT. Trong các thủ tục này, ta chấp nhận hai quy ước:

- Gốc của cây B luôn nằm trong bộ nhớ chính, để không bao giờ cần thực hiện một DISK-READ trên gốc; tuy nhiên, ta cần một DISK-WRITE của gốc, mỗi khi nút gốc thay đổi.
- Mọi nút được chuyển dưới dạng các tham số phải có sẵn một phép toán DISK-READ được thực hiện trên chúng.

Tất cả các thủ tục mà ta trình bày đều là các thuật toán "một lượt" [one-pass] tiến hành đổ xuống từ gốc của cây, mà không phải quay lại.

Tìm trong một cây B

Việc tìm trong một cây B cũng giống như tìm trong một cây tìm nhị phân, ngoại trừ thay vì thực hiện một quyết định rẽ nhánh nhị phân, hoặc "hai-chiều," tại mỗi nút, ta thực hiện một quyết định rẽ nhánh đa chiều theo số lượng các con của nút. Nói chính xác hơn, tại mỗi nút trong x, ta thực hiện một quyết định rẽ nhánh $(n[x] + 1)$ chiều.

B-TREE-SEARCH là một phép tổng quát hóa đơn giản của thủ tục TREE-SEARCH được định nghĩa cho các cây tìm nhị phân. B-TREE-SEARCH chấp nhận dưới dạng nhập liệu một biến trỏ đến nút gốc x của một cây con và một khóa k để tìm kiếm trong cây con đó. Như vậy, lệnh gọi cấp trên cùng có dạng B-TREE-SEARCH $(root[T], k)$. Nếu k nằm trong cây B, B-TREE-SEARCH trả về cặp có thứ tự (y, i) bao gồm một nút y và một chỉ số i sao cho $key_i[y] = k$. Bằng không, giá trị NIL được trả về.

B-TREE-SEARCH(x, k)

1 $i \leftarrow 1$

2 **while** $i \leq n[x]$ và $k > key_i[x]$

3   **do** $i \leftarrow i + 1$

4 **if** $i \leq n[x]$ và $k = key_i[x]$

5   **then** (x, i)

6 **if** $leaf[x]$

7   **then return** NIL

8   **else** DISK-READ$(c_i[x])$

9   **return** B-TREE-SEARCH$(c_i[x], k)$

Dùng một thủ tục tìm tuyến tính, các dòng 1-3 tìm i nhỏ nhất sao cho $k \leq key_i[x]$, nếu không chúng ấn định i là $n[x]+ 1$. Các dòng 4-5 kiểm tra để xem giờ đây ta đã phát hiện khóa chưa, trả về nếu có. Các dòng 6-9 kết thúc đợt tìm kiếm không thành công (nếu x là một lá) hoặc đệ quy để tìm trong cây con thích hợp của x, sau khi thực hiện DISK-READ cần thiết trên con đó.

Hình 19.1 minh họa phép toán B-TREE-SEARCH; các nút tô bóng sáng được xem xét trong một đợt tìm kiếm khóa R.

Cũng như trong thủ tục TREE-SEARCH của các cây tìm nhị phân, các nút đụng gặp trong đệ quy sẽ hình thành một lộ trình đổ xuống từ gốc của cây. Do đó, số lượng các trang đĩa mà B-TREE-SEARCH truy

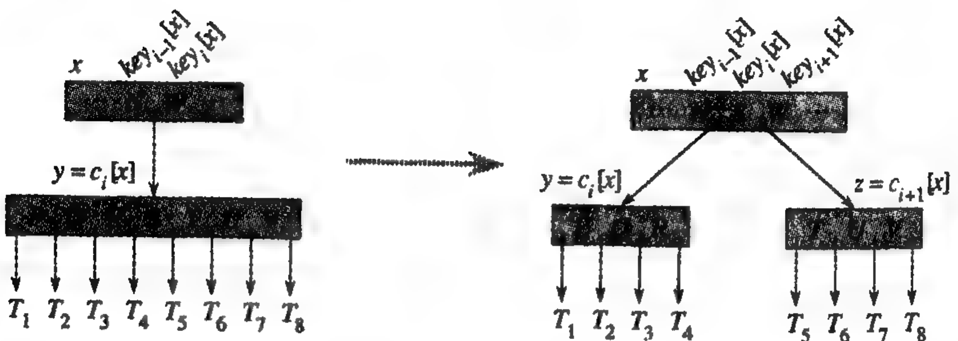
cập là $\Theta(h) = \Theta(\log_t n)$, ở đó h là chiều cao của cây B và n là số lượng khóa trong cây B. Do $n[x] < 2t$, nên thời gian mà vòng lặp **while** của các dòng 2-3 trải qua trong mỗi nút là $O(t)$, và tổng thời gian CPU là $O(th) = O(t \log_t n)$.

Tạo một cây B trống

Để xây dựng một cây B T, trước tiên ta dùng B-TREE-CREATE để tạo một nút gốc trống rồi gọi B-TREE-INSERT để bổ sung các khóa mới. Cả hai thủ tục này dùng một thủ tục phụ ALLOCATE-NODE, phân bổ một trang đĩa sẽ được dùng làm một nút mới trong $O(1)$ thời gian. Ta có thể mặc nhận một nút mà ALLOCATE-NODE tạo không yêu cầu DISK-READ, bởi cho đến lúc đó chưa có thông tin hữu ích nào được lưu trữ trên đĩa cho nút đó.

```
B-TREE-CREATE(T)
x ← ALLOCATE-NODE()
leaf[x] ← TRUE
n[x] ← 0
DISK-WRITE(x)
root[T] ← x
```

B-TREE-CREATE yêu cầu $O(1)$ phép toán đĩa và $O(1)$ thời gian CPU.



Hình 19.5 Tiến trình tách một nút có $t = 4$. Nút y được tách thành hai nút, y và z, và khóa trung tuyến S của y được dời lên vào cha của y.

Tách một nút trong một cây B

Tiến trình chèn một khóa vào một cây B thường phức tạp hơn nhiều so với việc chèn một khóa vào một cây tìm nhị phân. Một phép toán căn bản được dùng trong phép chèn đó là ***tách*** một nút đầy y (có 2t - 1

khóa) quanh ***khóa trung tuyến*** [median key] $key_i[y]$ của nó thành hai nút có t- 1 khóa cho mỗi nút. Khóa trung tuyến dời lên vào cha của y—phải không đầy trong khi tách y—để định danh điểm chia giữa hai cây mới; nếu y không có cha, thì cây tăng trưởng theo chiều cao lên một. Như vậy, tiến trình tách là biện pháp để cây tăng trưởng.

Thủ tục B-TREE-SPLIT-CHILD chấp nhận dưới dạng nhập liệu một nút *không đầy* trong x (mặc nhận nằm trong bộ nhớ chính), một chỉ số i, và một nút y sao cho $y = c_i[x]$ là một con đầy của x. Như vậy, thủ tục tách con này thành hai và dic x sao cho giờ đây nó có một con bổ sung.

Hình 19.5 minh họa tiến trình này. Nút đầy y được tách quanh khóa trung tuyến S của nó, được dời lên vào nút cha x của y. Những khóa trong y lớn hơn khóa trung tuyến đều được đặt trong một nút mới z, được tạo làm một con mới của x.

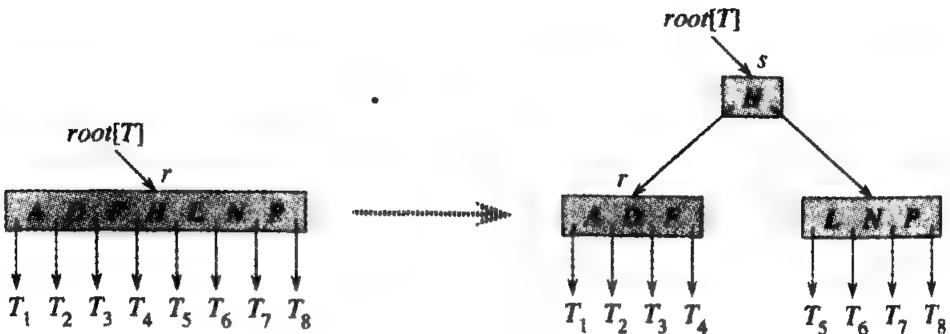
```
B-TREE-SPLIT-CHILD(x, i, y)
z ← ALLOCATE-NODE()
leaf[z] ← leaf[y]
n[z] ← t - 1
for j ← 1 to t - 1
     do key_j[z] ← key_{j+1}[y]
if not leaf[y]
     then for j ← 1 to t
               do c_j[z] - c_{j+1}[y]
n[y] ← t -1
for j ← n[x] + 1 downto i + 1
     do c_{j+1}[x] ← c_j[x]
c_{i+1}[x] ← z
for j ← n[x] downto i
     do key_{j+1}[x] ← key_j[x]
key_i[x] ← key_i[y]
n[x] ← n[x] + 1
DISK-WRITE(y)
DISK-WRITE(z)
DISK-WRITE(x)
```

B-TREE-SPLIT-CHILD làm việc bằng cách đơn giản "cắt và dán." Ở đây, y là con thứ i của x và là nút đang được tách. Ban đầu nút y có $2t$ - 1 con nhưng được rút gọn thành t - 1 con bởi phép toán này. Nút z "nhận làm con" t -1 con lớn nhất của y, và z trở thành một con mới của x, được định vị ngay sau y trong bảng các con của x. Khóa trung tuyến của y dời lên để trở thành khóa trong x tách y và z.

Các dòng 1-8 tạo nút z và cho nó t-1 khóa lớn hơn và t con tương ứng của y. Dòng 9 điều chỉnh số đếm khóa của y. Cuối cùng, các dòng 10-16 chèn z dưới dạng một con của x, dời khóa trung tuyến từ y lên đến x để tách biệt y với z, và điều chỉnh số khóa của x. Các dòng 17-19 ghi ra tất cả các trang đĩa đã sửa đổi. Thời gian CPU được B-TREE-SPLIT-CHILD sử dụng là $\Theta(t)$, do các vòng lặp trên các dòng 4-5 và 7-8. (Các vòng lặp khác chạy tối đa trong vòng t lần lặp lại.)

Chèn một khóa vào một cây B

Tiến trình chèn một khóa k vào một cây B T có chiều cao h được thực hiện trong một chỉ lượt [single pass] đổ xuống cây, yêu cầu $O(h)$ lần truy cập đĩa. Thời gian CPU cần thiết là $O(th) = O(t \log_t n)$. Thủ tục B-TREE-INSERT sử dụng B-TREE-SPLIT-CHILD để bảo đảm đệ quy không bao giờ hạ đến một nút đầy.



Hình 19.6 Tách gốc có t = 4. Nút gốc r được tách thành hai, và một gốc nút mới s được tạo. Gốc mới chứa khóa trung tuyến của r và có hai nửa của r làm các con. Cây B tăng trưởng theo chiều cao lên một khi gốc được tách.

```
B-TREE-INSERT(T, k)
    r ← root[T]
    if n[r] = 2t – 1
        then s ← ALLOCATE-NODE()
```

```
4           root[T] ← s
5           leaf[s] ← FALSE
6           n[s] ← 0
7           c1[s] ← r
8           B-TREE-SPLIT-CHILD(s, 1, r)
9           B-TREE-INSERT-NONFULL(s, k)
10 else B-TREE-INSERT-NONFULL(r, k)
```

Các dòng 3-9 điều quản trường hợp ở đó nút gốc *r* đầy: gốc được tách và một nút mới *s* (có hai con) trở thành gốc. Việc tách gốc là cách duy nhất để gia tăng chiều cao của một cây B. Hình 19.6 minh họa trường hợp này. Khác với một cây tìm nhị phân, một cây B gia tăng chiều cao trên đầu thay vì ở cuối. Thủ tục hoàn tất bằng cách gọi B-TREE-INSERT-NONFULL để thực hiện phép chèn khóa *k* trong cây có gốc tại nút gốc không đầy. Nếu cần, B-TREE-INSERT-NONFULL đệ quy đổ xuống cây, luôn bảo đảm rằng nút mà nó đệ quy theo sẽ không đầy bằng cách gọi B-TREE-SPLIT-CHILD nếu cần.

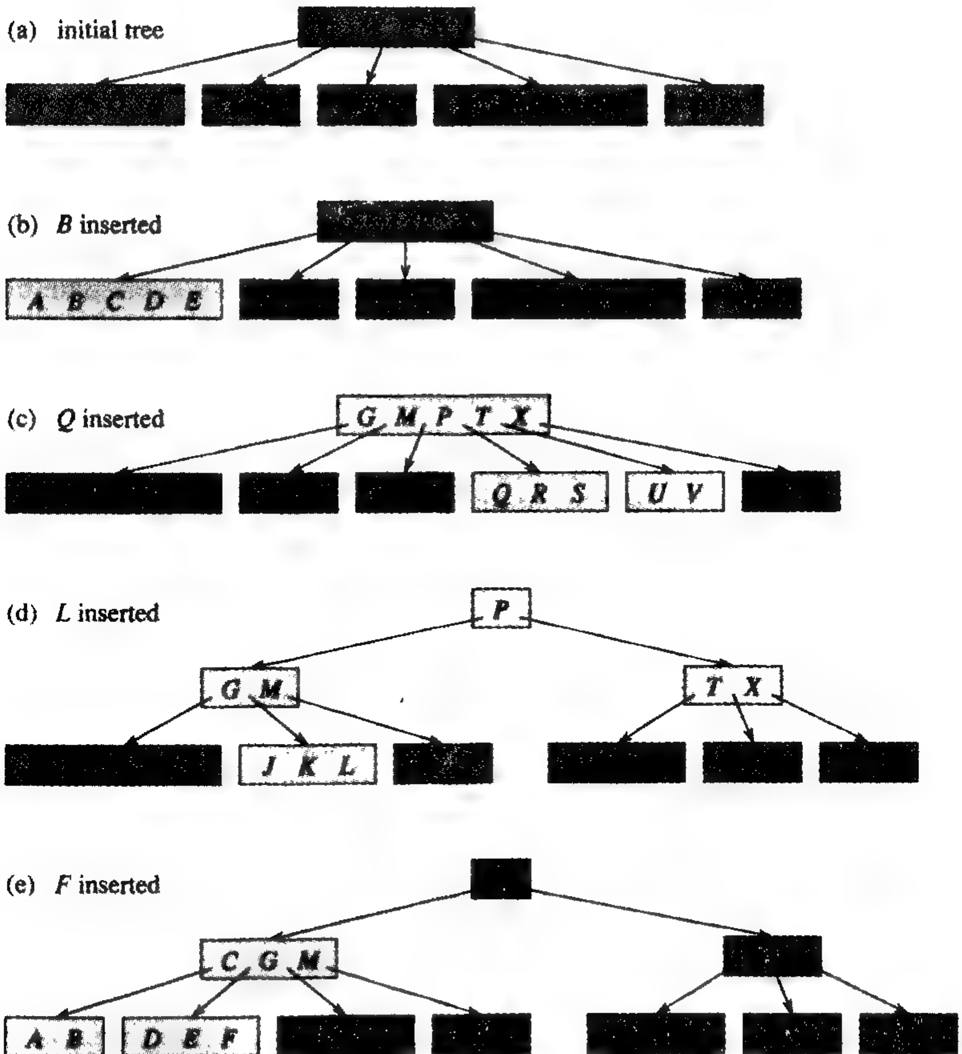
Thủ tục đệ quy phụ B-TREE-INSERT-NONFULL chèn khóa *k* vào nút *x*, được mặc nhận là không đầy khi thủ tục được gọi. Phép toán của B-TREE-INSERT và phép toán đệ quy của B-TREE-INSERT-NONFULL bảo đảm giả thiết này là đúng.

```
B-TREE-INSERT-NONFULL(x, k)
i ← n[x]
if leaf[x]
    then while i ≥ 1 và k < key_i[x]
            do key_{i+1}[x] ← key_i[x]
               i ← i - 1
        key_{i+1}[x] ← k
        n[x] ← n[x] + 1
        DISK-WRITE(x)
    else while i ≥ 1 và k < key_i[x]
            do i ← i - 1
        i ← i + 1
```

```
12          DISK-1ZEAD(c_i[x])
13          if n[c_i[x]] = 2t - 1
14              then B-TREE-SPLIT-CHILD(x, i, c_i[x])
15                  if k > key_i[x]
16                      then i ← i + 1
17          B-TREE-INSERT-NONFULL(c_i[x], k)
```

Thủ tục B-TREE-INSERT-NONFULL làm việc như sau. Các dòng 3-8 điều quản trường hợp ở đó x là một nút lá bằng cách chèn khóa k vào x. Nếu x không phải là nút lá, thì ta phải chèn k vào nút lá thích hợp trong cây con có gốc tại nút trong x. Trong trường hợp này, các dòng 9-11 xác định con của x nơi mà đệ quy đi xuống. Dòng 13 phát hiện xem đệ quy có đi xuống một con đầy hay không, trong trường hợp đó dòng 14 sử dụng B-TREE-SPLIT-CHILD để tách con đó thành hai con không đầy, và các dòng 13-16 xác định con nào trong số hai con giờ đây đúng là con để đi xuống. (Lưu ý, một DISK-READ($c_i[x]$) sau dòng 16 không cần gia số i, bởi trong trường hợp này phép đệ quy sẽ đi xuống một con vừa được B-TREE-SPLIT-CHILD tạo.) Như vậy, hiệu ứng chung cuộc của các dòng 13-16 đó là bảo đảm thủ tục không bao giờ đệ quy đến một nút đầy. Sau đó, dòng 17 đệ quy để chèn k vào cây con thích hợp. Hình 19.7 minh họa nhiều trường hợp khác nhau để chèn vào một cây B.

Số lần truy cập đĩa được thực hiện bởi B-TREE-INSERT là $O(h)$ cho một cây B có chiều cao h, bởi chỉ $O(1)$ phép toán DISK-READ và DISK-WRITE được thực hiện giữa các lệnh gọi đến B-TREE-INSERT-NONFULL. Tổng thời gian CPU được dùng là $O(th) = O(t \log_t n)$. Do B-TREE-INSERT-NONFULL là đệ quy-đuôi, nên ta có thể thực thi nó theo cách khác dưới dạng một vòng lặp **while**, chứng minh số lượng trang cần có trong bộ nhớ chính tại một thời điểm bất kỳ là $O(1)$.



Hình 19.7 Chèn các khóa vào một cây B. Độ cực tiểu t cho cây B này là 3, do đó một nút có thể lưu giữ tối đa 5 khóa. Các nút được sửa đổi bởi tiến trình chèn được tô bóng sáng. **(a)** Cây ban đầu cho ví dụ này. **(b)** Kết quả của việc chèn B vào cây ban đầu; đây là một phép chèn đơn giản vào một nút lá. **(c)** Kết quả của tiến trình chèn Q vào cây trước đó. Nút $RSTUV$ được tách thành hai nút chứa RS và UV, khóa T được dời lên đến gốc, và Q được chèn vào nút trái của hai nửa (nút RS). **(d)** Kết quả của việc chèn L vào cây trước đó. Gốc được tách ngay, bởi nó đầy, và cây B tăng trưởng theo chiều cao lên một. Thì L được chèn vào lá chứa JK. **(e)** Kết quả của việc chèn F vào cây trước đó. Nút $ABCDE$ được tách trước khi F được chèn vào nút phải của hai nửa (nút DE).

Bài tập

19.2-1

Nêu các kết quả của tiến trình chèn các khóa

F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E

theo thứ tự vào một cây B trống. Chỉ vẽ các cấu hình của cây ngay trước vài nút phải tách, và cũng vẽ cấu hình chung cuộc.

19.2-2

Giải thích dưới những hoàn cảnh nào, nếu có, các phép toán DISK-READ hoặc DISKWRITE dôi được thực hiện trong khi thi hành một lệnh gọi đến B-TREE-INSERT. (Một DISK-READ dôi là một DISK-READ về một trang đã có sẵn trong bộ nhớ. Một DISK-WRITE dôi ghi ra đĩa một trang thông tin giống như trang đã được lưu trữ sẵn ở đó.)

19.2-3

Giải thích cách tìm khóa cực tiểu được lưu trữ trong một cây B và cách tìm phần tử tiền vị của một khóa đã cho được lưu trữ trong một cây B.

19.2-4 *

Giả sử rằng các khóa $\{1, 2, ..., n\}$ được chèn vào một cây B trống có độ cực tiểu 2. Cây B chung cuộc có bao nhiêu nút?

19.2-5

Do các nút lá không yêu cầu biến trỏ nào đến các con, nên có thể tin rằng chúng có thể dùng một giá trị t khác (lớn hơn) ngoài các nút trong cho cùng kích cỡ trang đĩa. Nêu cách sửa đổi các thủ tục để tạo và chèn vào một cây B để điều quản biến thể này.

19.2-6

Giả sử B-TREE-SEARCH được thực thi để sử dụng kiểu tìm nhị phân thay vì tìm tuyến tính trong mỗi nút. Chứng tỏ điều này tạo thời gian CPU cần thiết $O(\lg n)$, độc lập với cách có thể chọn t làm một hàm của n.

19.2-7

Giả sử phần cứng đĩa cho phép ta chọn kích cỡ trang đĩa một cách tùy ý, song thời gian nó sửa đổi để đọc trang đĩa là $a + bt$, ở đó a và b là các hằng đã chỉ định và t là độ cực tiểu để một cây B dùng các trang có kích cỡ đã lựa. Mô tả cách chọn t để giảm thiểu (xấp xỉ) thời gian tìm kiếm cây B. Gợi ý một giá trị tối ưu của t cho trường hợp ở đó $a = 30$ miligiây và $b = 40$ microgiây.

19.3 Xóa một khóa ra khỏi một cây B

Tiến trình xóa ra khỏi một cây B cũng tương tự như phép chèn nhưng có phần phức tạp hơn. Ta phác họa cách làm việc của nó thay vì trình bày mã giả hoàn thành.

Giả sử thủ tục B-TREE-DELETE được yêu cầu xóa khóa *k* ra khỏi cây con có gốc tại *x*. Thủ tục này được cấu trúc để bảo đảm mỗi khi B-TREE-DELETE được gọi một cách đệ quy trên một nút *x*, số lượng khóa trong *x* ít nhất bằng độ cực tiểu *t*. Lưu ý, điều kiện này yêu cầu nhiều hơn một khóa so với mức cực tiểu mà các điều kiện cây B bình thường yêu cầu, sao cho thỉnh thoảng có thể phải dời một khóa vào một nút con trước khi đệ quy đi xuống con đó. Điều kiện gia cường này cho phép ta xóa một khóa ra khỏi cây trong một lượt đổ xuống mà không phải "quay lại" (với một ngoại lệ, mà ta sẽ giải thích sau). Phần chỉ định dưới đây cho phép xóa ra khỏi một cây B sẽ được diễn dịch với điều kiện nếu xảy ra trường hợp nút gốc *x* trở thành một nút trong không có khóa nào, thì *x* được xóa và con duy nhất $c_1[x]$ của *x* sẽ trở thành gốc mới của cây, giảm bớt chiều cao của cây là một và bảo toàn tính chất mà gốc của cây chứa ít nhất một khóa (trừ phi cây trống).

Hình 19.8 minh họa các trường hợp khác nhau của việc xóa các khóa ra khỏi một cây B.

1. Nếu khóa *k* nằm trong nút *x* và *x* là một lá, xóa khóa *k* ra khỏi *x*.

2. Nếu khóa *k* nằm trong nút *x* và *x* là một nút trong, thực hiện như sau.

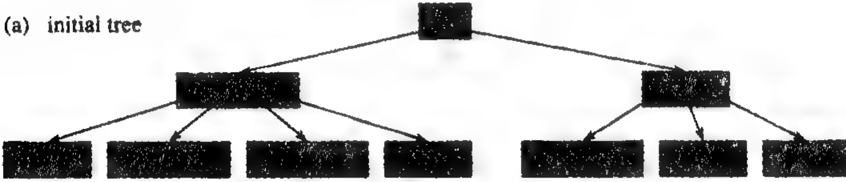
a. Nếu con *y* đứng trước *k* trong nút *x* có ít nhất *t* khóa, ta tìm phần tử tiền vị *k*' của *k* trong cây con có gốc tại *y*. Theo đệ quy xóa *k*', và thay *k* bằng *k*' trong *x*. (Có thể tiến hành tìm *k*' và xóa nó trong chỉ một lượt đổ xuống.)

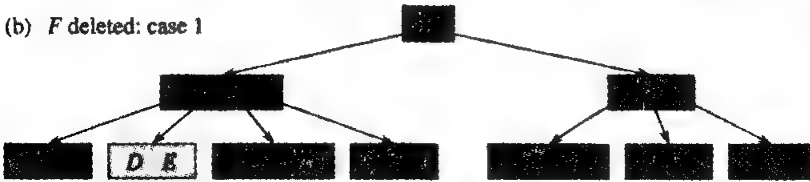
b. Theo đối xứng, nếu con *z* theo *k* trong nút *x* có ít nhất *t* khóa, hãy tìm phần tử kế vị *k*' của *k* trong cây con có gốc tại *z*. Theo đệ quy xóa *k*', và thay *k* bằng *k*' trong *x*. (Có thể tiến hành tìm *k*' và xóa nó trong chỉ một lượt đổ xuống.)

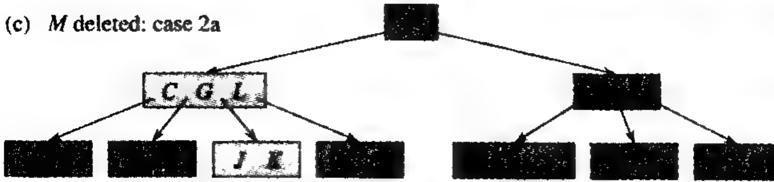
c. Bằng không, nếu cả *y* lẫn *z* chỉ có $t - 1$ khóa, hãy trộn *k* và tất cả *z* vào *y*, sao cho *x* bỏ qua cả *k* lẫn biến trỏ đến *z*, và giờ đây *y* chứa $2t - 1$ khóa. Sau đó, giải phóng *z* và xóa *k* ra khỏi *y* một cách đệ quy.

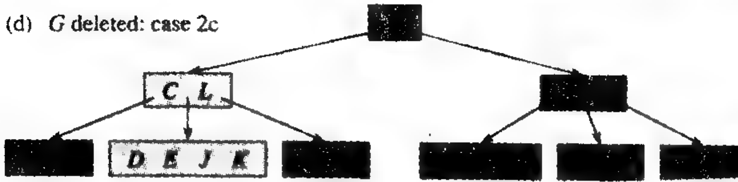
3. Nếu khóa *k* không hiện diện trong nút trong *x*, hãy xác định gốc $c_i[x]$ của cây con thích hợp phải chứa *k*, nếu *k* nằm trong cây. Nếu $c_i[x]$ chỉ có $t - 1$ khóa, thi hành bước 3a hoặc 3b khi cần để bảo đảm ta đi xuống đến một nút chứa ít nhất *t* khóa. Sau đó, hoàn tất bằng cách đệ quy trên con thích hợp của *x*.

a. Nếu $c_i[x]$ chỉ có t - 1 khóa nhưng có một anh em ruột với t khóa, hãy gán cho $c_i[x]$ một khóa phụ trội bằng cách dời một khóa từ x xuống vào $c_i[x]$, dời một khóa từ anh em ruột trái hoặc phải trực tiếp của $c_i[x]$ lên vào x, và dời con thích hợp từ anh em ruột vào $c_i[x]$.

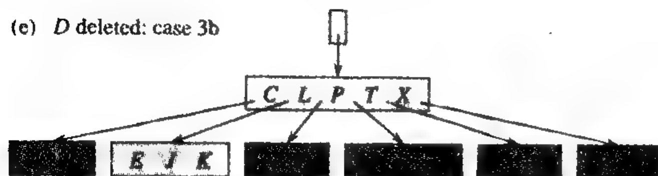








Hình 19.8 Xóa các khóa ra khỏi một cây B. Độ cực tiểu của cây B này là t = 3, do đó một nút (ngoài gốc) không thể có ít hơn 2 khóa. Các nút được sửa đổi tô bóng sáng. **(a)** Cây B của Hình 19.7(*e*). **(b)** Xóa *F*. Đây là trường hợp 1: phép xóa đơn giản ra khỏi một lá. **(c)** Xóa *M*. Đây là trường hợp 2a: phần tử tiền vị *L* của *M* được dời lên để chiếm vị trí của *M*. **(d)** Xóa *G*. Đây là trường hợp 2c: *G* được đẩy xuống để tạo nút *DEGJK*, rồi *G* được xóa ra khỏi lá này (trường hợp 1). (*e*) Xóa *D*. Đây là trường hợp 3b: đệ quy không thể đi xuống đến nút *CL* bởi nó chỉ có 2 khóa, do đó *P* được đẩy xuống và hợp nhất với *CL* và *TX* để tạo thành *CLPTX;* sau đó, *D* được xóa ra khỏi một lá (trường hợp 1). (*e'*) Sau (d), gốc được xóa và cây có cụm chiều cao theo một. (*f*) Xóa *B*. Đây là trường hợp 3a: *C* được dời để điền vị trí của *B* và *E* được dời để điền vị trí của *C*.







b. Nếu $c_i[x]$ và tất cả các anh em ruột của $c_i[x]$ có t-1 khóa, hãy trộn c_i với một anh em ruột, kéo theo việc di chuyển một khóa từ x xuống vào nút mới hợp nhất để trở thành khóa trung tuyến cho nút đó.

Bởi hầu hết các khóa trong một cây B nằm trong các lá, ta có thể dự kiến rằng trong thực tế, các phép toán xóa thường được dùng nhất để xóa các khóa ra khỏi các lá. Như vậy, thủ tục B-TREE-DELETE tác động theo một lượt đổ xuống qua cây, mà không phải quay lại. Tuy nhiên, khi xóa một khóa trong một nút trong, thủ tục thực hiện một lượt đổ xuống qua cây nhưng có thể phải trở về nút mà khóa đã được xóa ra khỏi đó để thay khóa bằng phần tử tiền vị hoặc phần tử kế vị của nó (các trường hợp 2a và 2b).

Mặc dù thủ tục này có vẻ phức tạp, nó chỉ liên quan đến $O(h)$ phép toán đĩa cho một cây B có chiều cao h, bởi chỉ có $O(1)$ lệnh gọi đến DISK-READ và DISK-WRITE giữa các lần triệu gọi đệ quy của thủ tục. Thời gian CPU cần có là $O(th) = O(t \log_t n)$.

Bài tập

19.3-1

Nêu các kết quả của việc xóa C, P, và V, trong thứ tự, ra khỏi cây trong Hình 19.8(f).

19.3-2

Viết mã giả cho B-TREE-DELETE.

Các Bài Toán

19-1 Các ngăn xếp trên kho lưu trữ thứ cấp

Xét cách thực thi một ngăn xếp trong một máy tính có một lượng bộ nhớ chính nhanh tương đối nhỏ và một lượng kho lưu trữ đĩa chậm hơn tương đối lớn. Các phép toán PUSH và POP được hỗ trợ trên các giá trị từ đơn [single-word]. Ngăn xếp mà ta muốn hỗ trợ có thể tăng trưởng trở nên lớn hơn nhiều so với khả năng sẵn dùng trong bộ nhớ, và như vậy hầu hết nó phải được lưu trữ trên đĩa.

Một cách thực thi ngăn xếp tuy đơn giản, song không hiệu quả đó là lưu giữ nguyên cả ngăn xếp trên đĩa. Ta duy trì trong bộ nhớ một biến trỏ ngăn xếp, là địa chỉ đĩa của thành phần trên cùng trên ngăn xếp. Nếu biến trỏ có giá trị p, thành phần trên cùng là từ thứ (p mod m) trên trang $\lfloor p/m \rfloor$ của đĩa, ở đó m là số lượng từ mỗi trang.

Để thực thi phép toán PUSH, ta gia số biến trỏ ngăn xếp, đọc trang thích hợp vào bộ nhớ từ đĩa, chép thành phần sẽ được đẩy đến từ thích hợp trên trang, và viết trang trở lại đĩa. Một phép toán POP cũng tương tự. Ta giảm lượng biến trỏ ngăn xếp, đọc vào trang thích hợp từ đĩa, và trả về đỉnh của ngăn xếp. Ta không cần ghi trang trở lại, bởi nó không bị sửa đổi.

Bởi các phép toán đĩa tương đối tốn kém, nên ta dùng tổng số lần truy cập đĩa như một số liệu khen thưởng với bất kỳ kiểu thực thi nào. Ta cũng tính thời gian CPU, nhưng tính công $\Theta(m)$ cho bất kỳ lần truy cập đĩa nào đến một trang gồm m từ.

a. Theo tiệm cận, đâu là số lần truy cập đĩa ca xấu nhất cho n phép toán ngăn xếp dùng cách thực thi đơn giản này? Đâu là thời gian CPU cho n phép toán ngăn xếp? (Biểu diễn đáp án của bạn theo dạng m và n cho phần này và các phần tiếp theo.)

Giờ đây, hãy xét một kiểu thực thi ngăn xếp ở đó ta duy trì một trang của ngăn xếp trong bộ nhớ. (Ta cũng duy trì một lượng bộ nhớ nhỏ để theo dõi trang nào hiện nằm trong bộ nhớ.) Ta chỉ có thể thực hiện một phép toán ngăn xếp nếu trang đĩa có liên quan thường trú trong bộ nhớ. Nếu cần, trang hiện nằm trong bộ nhớ có thể được ghi ra đĩa và ghi trang mới đọc từ đĩa ra bộ nhớ. Nếu trang đĩa có liên quan hiện nằm trong bộ nhớ, thì không cần truy cập đĩa.

b. Đâu là số lần truy cập đĩa ca xấu nhất cần có cho n phép toánPUSH? Đâu là thời gian CPU?

c. Đâu là số lần truy cập đĩa ca xấu nhất cần có cho n phép toán ngăn xếp? Đâu là thời gian CPU?

Giả sử rằng giờ đây ta thực thi ngăn xếp bằng cách giữ hai trang trong bộ nhớ (ngoài một số lượng nhỏ của từ để theo dõi).

d. Mô tả cách quản lý các trang ngăn xếp sao cho số lần truy cập đĩa khấu trừ của bất kỳ phép toán ngăn xếp nào là $O(1/m)$ và thời gian CPU khấu trừ của bất kỳ phép toán ngăn xếp nào là $O(1)$.

19-2 Ghép và tách các cây 2-3-4

Phép toán ghép lấy hai tập hợp động S' và S'' và một thành phần x sao cho với bất kỳ $x' \in S'$ và $x'' \in S''$, ta có $key[x'] < key[x] < key[x'']$. Nó trả về một tập hợp $S = S' \cup \{x\} \cup S''$. Phép toán ***tách*** cũng giống như một phép ghép "nghịch đảo": cho một tập hợp động S và một thành phần $x \in S$, nó tạo một tập hợp S' bao gồm tất cả các thành phần trong $S - \{x\}$ có các khóa nhỏ hơn $key[x]$ và một tập hợp S'' bao gồm tất cả các thành phần trong $S - \{x\}$ có các khóa lớn hơn $key[x]$. Trong bài toán này, ta nghiên cứu cách thực thi các phép toán này trên các cây 2-3-4. Để tiện dụng, ta mặc nhận các thành phần chỉ bao gồm các khóa và tất cả các giá trị khóa là riêng biệt.

a. Với mọi nút x của một cây 2-3-4, hãy nêu cách duy trì chiều cao của cây con có gốc tại x dưới dạng một trường *height*$[x]$. Bảo đảm cách thực thi của bạn không ảnh hưởng đến các thời gian thực hiện tiệm cận để tìm, chèn, và xóa.

b. Nêu cách thực thi phép toán ghép. Cho hai cây 2-3-4 T' và T'' và một khóa k, phép ghép sẽ chạy trong $O(|h' - h''|)$ thời gian, ở đó h' và h'' là các chiều cao của T' và T'', theo thứ tự nêu trên.

c. Xét lộ trình p từ gốc của một cây 2-3-4 T đến một khóa k đã cho, tập hợp S' các khóa trong T nhỏ hơn k, và tập hợp S'' các khóa trong T lớn hơn k. Chứng tỏ p tách S' thành một tập hợp các cây $\{T'_0, T'_1, \ldots, T'_m\}$ và một tập hợp các khóa $\{k'_1, k'_2, \ldots, k'_m\}$, ở đó, với $i = 1, 2, \ldots, m$, ta có $y < k'_i < z$ với bất kỳ các khóa $y \in T'_{i-1}$ và $z \in T'_i$. Đâu là mối quan hệ giữa các chiều cao của T'_{i-1} và T'_i? Mô tả cách p tách S'' thành các tập hợp các cây và các khóa.

d. Nêu cách thực thi phép toán tách trên T. Dùng phép toán ghép để lắp ráp các khóa trong S' thành một cây 2-3-4 T' đơn lẻ và các khóa trong S'' thành một cây 2-3-4 T'' đơn lẻ. Thời gian thực hiện của phép toán tách sẽ là $O(\lg n)$, ở đó n là số lượng khóa trong T. (*Mách nước:* Các mức hao phí để ghép phải thu gọn.)

Ghi chú Chương

Knuth [123], Aho, Hopcroft, và Ullman [4], và Sedgewick [175] đề cập kỹ hơn về các lược đồ cây cân đối và các cây B. Comer [48] cung cấp một nghiên cứu toàn diện về các cây B. Guibas và Sedgewick [93] đề cập các mối quan hệ giữa các kiểu lược đồ cây cân đối khác nhau, kể cả các cây đỏ đen và các cây 2-3-4.

Vào năm 1970, J. E. Hopcroft đã phát minh các cây2-3, tiền thân của các cây B và các cây 2-3-4, ở đó mọi nút trong đều có hai hoặc ba con. Các cây B đã được Bayer và McCreight giới thiệu vào năm 1972 [18]; họ không giải thích cách chọn tên của họ.

20 Các Đống Nhị Thức

Chương này và Chương 21 trình bày các cấu trúc dữ liệu có tên là ***các đống khả trộn*** [mergeable heaps], hỗ trợ năm phép toán dưới đây.

MAKE-HEAP() tạo và trả về một đống mới không chứa thành phần nào.

INSERT(H, x) chèn nút x, có trường *key* đã điền sẵn, vào đống H.

MINIMUM(H) trả về một biến trỏ đến nút trong đống H có khóa là cực tiểu.

EXTRACT-MIN(H) xóa nút ra khỏi đống H có khóa là cực tiểu, trả về một biến trỏ đến nút.

UNION (H_1, H_2) tạo và trả về một đống mới chứa tất cả các nút của các đống H_1 và H_2. Các đống H_1 và H_2 bị "hủy" bởi phép toán này.

Ngoài ra, các cấu trúc dữ liệu trong các chương này cũng hỗ trợ hai phép toán dưới đây.

DECREASE-KEY(H, x, k) gán cho nút x trong đống H giá trị khóa mới k, được mặc nhận là không lớn hơn giá trị khóa hiện hành của nó.

DELETE(H, x) xóa nút x ra khỏi đống H.

Như bảng trong Hình 20.1 đã nêu, nếu ta không cần phép toán UNION, các đống nhị phân bình thường cũng làm việc tốt, như từng được dùng trong kỹ thuật sắp xếp đống (Chương 7). Các phép toán khác ngoài UNION chạy trong thời gian trường hợp xấu nhất $O(\lg n)$ (hoặc tốt hơn) trên một đống nhị phân. Tuy nhiên, nếu phải hỗ trợ phép toán UNION, các đống nhị phân thực hiện khá tồi. Nhờ ghép nối hai mảng lưu giữ các đống nhị phân sẽ được hợp nhất rồi chạy HEAPIFY, phép toán UNION mất $\Theta(n)$ thời gian trong trường hợp xấu nhất.

Trong chương này, ta sẽ xét "các đống nhị thức," mà các cận thời gian trường hợp xấu nhất của chúng cũng được nêu trong Hình 20.1. Nói cụ thể, phép toán UNION chỉ mất $O(\lg n)$ thời gian để trộn hai đống nhị thức với một tổng n thành phần.

Trong Chương 21, ta sẽ khảo sát các đống Fibonacci, có thậm chí các cận thời gian tốt hơn cho vài phép toán. Tuy nhiên, lưu ý, các thời gian

thực hiện của các đống Fibonacci trong Hình 20.1 là các cận thời gian khấu trừ, chứ không phải là các cận thời gian trường hợp xấu nhất của mỗi phép toán.

Thủ tục	Đống nhị phân (trường hợp xấu nhất)	Đống nhị thức (trường hợp xấu nhất)	Đống Fibonacci (khấu trừ)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$O(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$O(\lg n)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$O(\lg n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$

Hình 20.1 Các thời gian thực hiện của các phép toán trên ba kiểu thực thi các đống khả trộn. Số lượng các mục trong (các) đống tại thời gian của một phép toán được nêu rõ bởi n.

Chương này bỏ qua các vấn đề như phân bổ các nút trước phép chèn và giải phóng các nút sau phép xóa. Ta mặc nhận mã gọi các thủ tục đống sẽ điều quản các chi tiết này.

Các đống nhị phân, các đống nhị thức, và các đống Fibonacci, tất cả đều không hiệu quả trong việc hỗ trợ phép toán SEARCH của chúng; có thể phải mất một thời gian để tìm ra một nút có một khóa đã cho. Vì lý do này, các phép toán như DECREASE-KEY và DELETE tham chiếu một nút đã cho sẽ yêu cầu một biến trỏ đến nút đó dưới dạng một phần nhập liệu của chúng. Yêu cầu này không thành vấn đề trong nhiều ứng dụng.

Đoạn 20.1 định nghĩa các đống nhị thức sau khi định nghĩa lần đầu tiên các cây nhị thức cấu thành của chúng. Nó cũng giới thiệu một phần biểu diễn cụ thể về các đống nhị thức. Đoạn 20.2 nêu cách thực thi các phép toán trên các đống nhị thức trong các cận thời gian đã cho trong Hình 20.1.

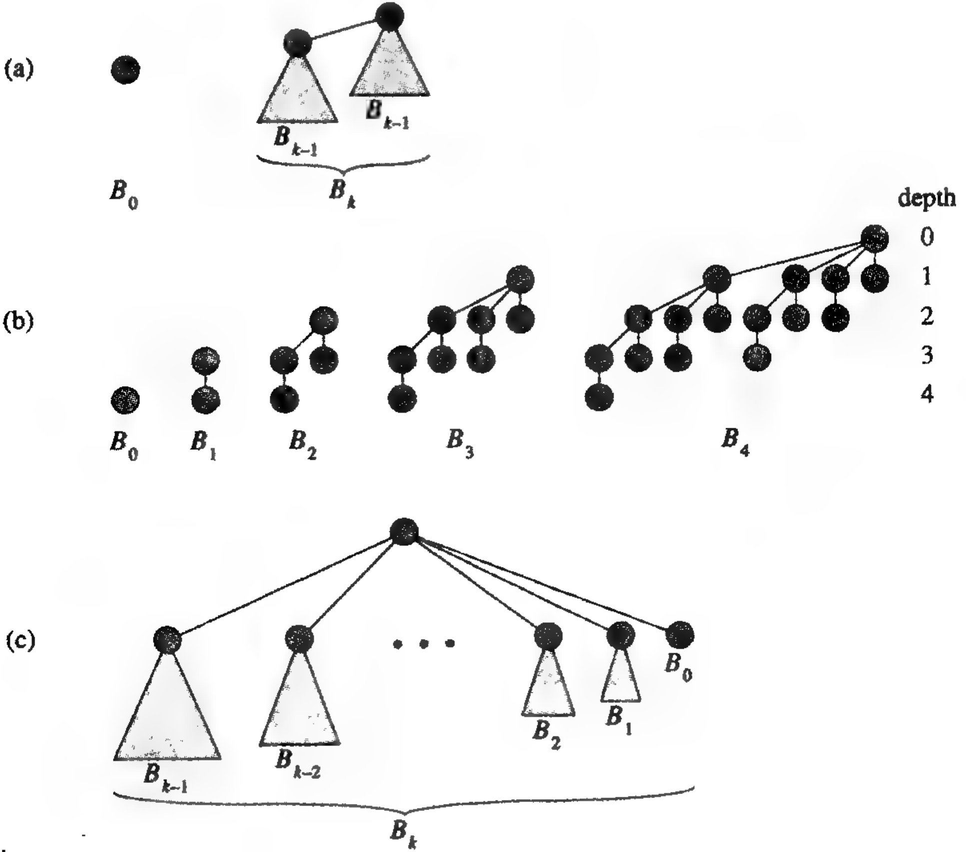
20.1 Các cây nhị thức và các đống nhị thức

Một đống nhị thức [binomial heap] là một tập hợp các cây nhị thức, do đó đoạn này bắt đầu bằng cách định nghĩa các cây nhị thức và chứng minh vài tính chất chính. Sau đó, ta định nghĩa các đống nhị thức và nêu cách biểu thị chúng.

20.1.1 Các cây nhị thức

Cây nhị thức B_k là một cây có thứ tự (xem Đoạn 5.5.2) được định nghĩa theo đệ quy. Như đã nêu trong Hình 20.2(a), cây nhị thức B_0 bao gồm một nút đơn lẻ. Cây nhị thức B_k bao gồm hai cây nhị thức B_{k-1} được ***nối kết*** với nhau: gốc của cây này là con mút trái của gốc cây kia. Hình 20.2(b) nêu các cây nhị thức B_0 đến B4

Bổ đề dưới đây có nêu vài tính chất của các cây nhị thức



Hình 20.2 (a) Phần định nghĩa đệ quy của cây nhị thức B_k. Các tam giác biểu thị cho các cây con có gốc. **(b)** Các cây nhị thức B_0 đến B_4. Các độ sâu nút trong B_4 được nêu. **(c)** Một cách khác để xem xét cây nhị thức B_k.

Bổ đề 20.1 (Các tính chất của các cây nhị thức)

Với cây nhị thức B_k,

1. có 2^k nút,

2. chiều cao của cây là k,

3. có chính xác $\binom{k}{i}$ nút tại độ sâu i với $i = 0, 1,..., k$, và

4. gốc có độ k, lớn hơn của bất kỳ nút nào khác; hơn nữa nếu các con của gốc được đánh số từ trái qua phải theo $k - 1, k - 2, \ldots, 0$, con i là gốc của một cây con B_i.

Chứng minh Để chứng minh, ta theo phương pháp quy nạp trên k. Với mỗi tính chất, cơ sở là cây nhị thức B_0. Xác minh mỗi tính chất áp dụng cho B_0 là không quan trọng.

Với bước quy nạp, ta mặc nhận bổ đề áp dụng cho B_{k-1}.

1. Cây nhị thức B_k bao gồm hai bản sao của B_{k-1}, do đó B_k có $2^{k-1} + 2^{k-1} = 2^k$ nút.

2. Do cách thức ở đó hai bản sao của B_{k-1} được nối kết để tạo thành *Bk*, nên độ sâu cực đại của một nút trong B_k chính là độ sâu lớn hơn độ sâu cực đại trong B_{k-1}. Theo giả thuyết quy nạp, độ sâu cực đại này là $(k - 1) + 1 = k$.

3. Cho $D(k, i)$ là số lượng các nút tại độ sâu i của cây nhị thức B_k. Bởi B_k bao gồm hai bản sao của B_{k-1} được nối kết với nhau, nên một nút tại độ sâu i trong B_{k-1} sẽ xuất hiện trong B_k một lần tại độ sâu i và một lần tại độ sâu $i + 1$. Nói cách khác, số lượng nút tại độ sâu i trong B_k chính là số lượng nút tại độ sâu i trong B_{k-1} cộng với số lượng nút tại độ sâu $i - 1$ trong B_{k-1}. Như vậy,

$$\begin{aligned} D(k, i) &= D(k-1, i) + D(k-1, i-1) \\ &= \binom{k-1}{i} + \binom{k-1}{i-1} \\ &= \binom{k}{i} . \end{aligned}$$

Đẳng thức thứ hai là do giả thuyết quy nạp, và đẳng thức thứ ba là do Bài tập 6.1-7.

4. Nút duy nhất có độ lớn hơn trong B_k so với trong B_{k-1} chính là gốc, có nhiều hơn một con so với B_{k-1}. Do gốc của B_{k-1} có độ $k - 1$, nên gốc của B_k có độ k. Giờ đây theo giả thuyết quy nạp, và như Hình 20.2(c) đã nêu, từ trái qua phải, các con của gốc B_{k-1} là các gốc của $B_{k-2}, B_{k-3}, \ldots, B_0$. Do đó, khi B_{k-1} được nối kết với B_{k-1}, các con của gốc kết quả chính là các gốc của $B_{k-1}, B_{k-2}, \ldots, B_0$.

Hệ luận 20.2

Độ cực đại của một nút bất kỳ trong một cây nhị thức n-nút sẽ là $\lg n$.

Chứng minh Tức thời từ các tính chất 1 và 4 của Bổ đề 20.1.

Thuật ngữ "cây nhị thức" [binomial tree] xuất xứ từ tính chất 3 của Bổ đề 20.1, bởi

các số hạng $\binom{k}{i}$ là các hệ số nhị thức. Bài tập 20.1-3 xác minh thêm về thuật ngữ này.

20.1.2 Các đống nhị thức

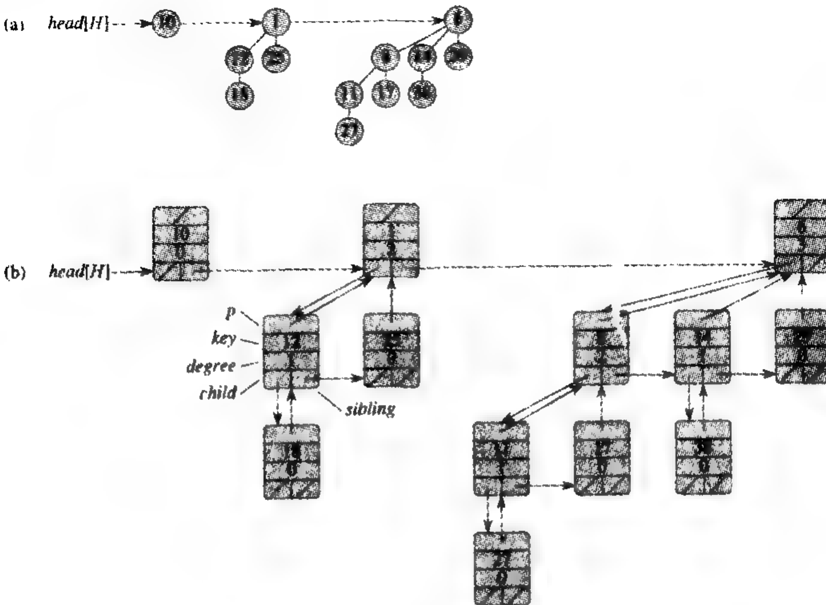
Một ***đống nhị thức*** [binomial heap] *H* là một tập hợp các cây nhị thức thỏa ***các tính chất đống nhị thức*** dưới đây.

1. Mỗi cây nhị thức trong *H* ***được sắp xếp theo đống***: khóa của một nút sẽ lớn hơn hoặc bằng với khóa của cha nó.

2. Có tối đa một cây nhị thức trong *H* mà gốc của nó có một độ đã cho.

Tính chất đầu tiên báo cho ta biết gốc của một cây được sắp xếp theo đống sẽ chứa khóa nhỏ nhất trong cây.

Tính chất thứ hai hàm ý một đống nhị thức *H* *n*-nút gộp tối đa $\lfloor \lg n \rfloor + 1$ cây nhị thức. Để biết tại sao, ta nhận thấy phần biểu diễn nhị phân của *n* có $\lfloor \lg n \rfloor + 1$ bit, giả sử $<b_{\lfloor \lg n \rfloor}, b_{\lfloor \lg n \rfloor} - 1, ..., b_0>$, sao cho $n = \sum_{i=0}^{\lfloor \lg n \rfloor} b_i 2^i$. Do đó, theo tính chất 1 của Bổ đề 20.1, cây nhị thức B_i xuất hiện trong *H* nếu và chỉ nếu bit $b_i = 1$. Như vậy, đống nhị thức *H* chứa tại hầu hết $\lfloor \lg n \rfloor + 1$ cây nhị thức.

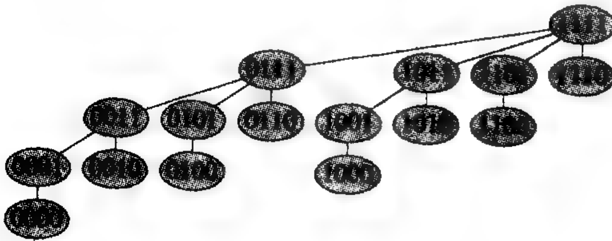


Hình 20.3 Một đống nhị thức *H* có $n = 13$ nút. **(a)** Đống bao gồm các cây nhị thức B_0, B_2, và B_3, có 1, 4, và 8 nút theo thứ tự nêu trên, tổng cộng $n = 13$ nút. Do mỗi cây nhị thức được sắp xếp theo đống, nên khóa của một nút bất kỳ sẽ không nhỏ hơn khóa của cha nó. Ở đây cũng nêuj danh sách gốc, là một danh sách nối kết các gốc theo thứ tự độ tăng dần. **(b)** Một phần biểu diễn chi tiết hơn của đống nhị thức *H*. Mỗi cây nhị thức được lưu trữ trong con trái, phần biểu diễn anh em ruột bên phải, và mỗi nút lưu trữ độ của nó.

Hình 20.3(a) nêu một đống nhị thức H có 13 nút. Phần biểu diễn nhị phân của 13 là <1101>, và H bao gồm các cây nhị thức được sắp xếp theo đống B_3, B_2, và B_0, có 8, 4, và 1 nút theo thứ tự nêu trên, với một tổng là 13 nút.

Biểu thị các đống nhị thức

Như đã nêu trong Hình 20.3(b), mỗi cây nhị thức trong một đống nhị thức được lưu trữ trong con trái, phần biểu diễn anh em ruột bên phải của Đoạn 11.4. Mỗi nút có một trường *key* và mọi thông tin vệ tinh khác mà ứng dụng yêu cầu. Ngoài ra, mỗi nút x chứa các biến trỏ $p[x]$ đến cha của nó, $child[x]$ đến con mút trái của nó, và $sibling[x]$ đến anh em ruột của x ngay bên phải của nó. Nếu nút x là một gốc, thì $p[x]$ = NIL. Nếu nút x không có con, thì $child[x]$ = NIL, và nếu x là con mút phải của cha nó, thì $sibling[x]$ = NIL. Mỗi nút x cũng chứa trường $degree[x]$, là số lượng con của x.



Hình 20.4 Cây nhị thức B_4 có các nút gắn nhãn theo nhị phân bởi một tầng có thứ tự sau [postorder walk].

Như Hình 20.3 đã nêu, các gốc của các cây nhị thức trong một đống nhị thức được tổ chức theo một danh sách nối kết, mà ta gọi là ***danh sách gốc***. Các độ của các gốc tăng ngặt khi ta băng ngang danh sách gốc. Theo tính chất đống nhị thức thứ hai, trong một đống nhị thức n-nút, các độ của các gốc là một tập hợp con $\{0, 1, \ldots, \lfloor \lg n \rfloor\}$. Với các gốc, trường *sibling* có một ý nghĩa khác so với không phải gốc. Nếu x là một gốc, thì $sibling[x]$ trỏ đến gốc kế tiếp trong danh sách gốc. (Như thường lệ, $sibling[x]$ = NIL nếu x là gốc cuối trong danh sách gốc.)

Một đống nhị thức đã cho H được trường $head[H]$ truy cập, đơn giản là một biến trỏ đến gốc đầu tiên trong danh sách gốc của H. Nếu đống nhị thức H không có thành phần nào, thì $head[H]$ = NIL.

Bài tập

20.1-1

Giả sử x là một nút trong một cây nhị thức trong một đống nhị thức,

và mặc nhận *sibling*[*x*] ≠ NIL. Nếu *x* không phải là một gốc, làm sao để so sánh *degree*[*sibling*[*x*]] với *degree*[*x*]? Sẽ là gì nếu *x* là một gốc?

20.1-2

Nếu *x* là một nút không phải gốc trong một cây nhị thức trong một đống nhị thức, *degree*[*p*[*x*]] sẽ so sánh với *degree*[*x*] như thế nào?

20.1-3

Giả sử ta gắn nhãn các nút của cây nhị thức B_k theo nhị phân bởi một duyệt hậu cấp [postorder walk], như trong Hình 20.4. Hãy xét một nút *x* có nhãn *l* tại độ sâu *i*, và cho $j = k - i$. Chứng tỏ *x* có *j* 1 trong phần biểu diễn nhị phân của nó. Có bao nhiêu chuỗi-*k* nhị phân chứa chính xác *j* 1? Chứng tỏ độ của *x* bằng với số lượng 1 về bên phải của 0 mút phải trong phần biểu diễn nhị phân của *l*.

20.2 Các phép toán trên các đống nhị thức

Trong đoạn này, ta nêu cách thực hiện các phép toán trên các đống nhị thức trong các cận thời gian được nêu trong Hình 20.1. Ta sẽ chỉ nêu các cận trên; các cận dưới được để lại làm Bài tập 20.2-10.

Tạo một đống nhị thức mới

Để tạo một đống nhị thức trống, thủ tục MAKE-BINOMIAL-HEAP đơn giản phân bổ và trả về một đối tượng *H*, ở đó *head*[*H*] = NIL. Thời gian thực hiện là Θ(1).

Tìm khóa cực tiểu

Thủ tục BINOMIAL-HEAP-MINIMUM trả về một biến trỏ đến nút có khóa cực tiểu trong một đống nhị thức *H* *n*-nút. Cách thực thi này mặc nhận không có các khóa có giá trị ∞. (Xem Bài tập 20.2-5.)

```
BINOMIAL-HEAP-MINIMUM(H)
y ← NIL
x ← head[H]
min ← ∞
while x ≠ NIL
      do if key[x] < min
               then min ← key[x]
                    y ← x
```

```
        x ← sibling[x]
return y
```

Bởi một đống nhị thức được sắp xếp theo đống, nên khóa cực tiểu phải thường trú trong một nút gốc. Thủ tục BINOMIAL-HEAP-MINIMUM kiểm tra tất cả các gốc, có số tối đa $\lfloor \lg n \rfloor + 1$, lưu khóa cực tiểu hiện hành trong *min* và một biến trỏ đến khóa cực tiểu hiện hành trong *y*. Khi triệu gọi đống nhị thức trong Hình 20.3, BINOMIAL-HEAP-MINIMUM trả về một biến trỏ đến nút có khóa 1.

Bởi có tối đa $\lfloor \lg n \rfloor + 1$ gốc để kiểm tra, nên thời gian thực hiện của BINOMIAL-HEAP-MINIMUM là $O(\lg n)$.

Hợp nhất hai đống nhị thức

Phép toán hợp nhất hai đống nhị thức được hầu hết các phép toán còn lại sử dụng dưới dạng một chương trình con. Thủ tục BINOMIAL-HEAP-UNION liên tục nối kết các cây nhị thức có các gốc có cùng độ. Thủ tục dưới đây nối kết cây B_{k-1}, có gốc tại nút *y* với cây B_{k-1} có gốc tại nút *z*; nghĩa là, nó khiến *z* trở thành cha của *y*. Như vậy, nút *z* trở thành gốc của một cây B_k.

```
BINOMIAL-LINK(y, z)
p[y] ← z
sibling[y] ← child[z]
child [z] ← y
degree[z] ← degree[z] + 1
```

Thủ tục BINOMIAL-LINK khiến nút *y* trở thành đầu mới của danh sách nối kết các con của nút *z* trong $O(1)$ thời gian. Nó làm việc bởi phần biểu diễn con trái, anh em ruột phải của mỗi cây nhị thức sẽ khớp với tính chất sắp xếp thứ tự của cây: trong một cây B_k, con mút trái của gốc chính là gốc của một cây B_{k-1}.

Thủ tục dưới đây hợp nhất các đống nhị thức H_1 và H_2, trả về đống kết quả. Nó hủy các phần biểu diễn của H_1 và H_2 trong tiến trình. Ngoài BINOMIAL-LINK, thủ tục sử dụng một thủ tục phụ BINOMIAL-HEAP-MERGE trộn các danh sách gốc của H_1 và H_2 thành một danh sách nối kết đơn lẻ được sắp xếp theo độ theo thứ tự tăng đơn điệu. Thủ tục BINOMIAL-HEAP-MERGE, mà ta chừa phần mã giả của nó làm Bài tập 20.2-2, tương tự như thủ tục MERGE trong Đoạn 1.3.1.

```
BINOMIAL-HEAP-UNION(H1, H2)
H ← MAKE-BINOMIAL-HEAP()
head[H] ← BINOMIAL-HEAP-MERGE(H1, H2)
giải phóng các đối tượng H1 và H2 nhưng không phải là các danh sách mà chúng trỏ đến
if head [H] = NIL
    then return H
prev-x ← NIL
x ← head[H]
next-x ← sibling[x]
while next-x ≠ NIL
    do if (degree[x] ≠ degree[next-x]) hoặc
                (sibling[next-x] ≠ NIL
                và degree[sibling[next-x]] = degree[x])
          then prev-x ← x ▷ Các trường hợp 1 và 2
               x ← next-x ▷ Các trường hợp 1 và 2
          else if key[x] ≤ key[next-x]
                 then sibling[x] ← sibling[next-x]
                                                ▷ Trường hợp 3
                      BINOMIAL-LINK(next-x, x)
                                                ▷ Trường hợp 3
               else if prev-x = NIL ▷ Trường hợp 4
                      then head [H] ← next-x    ▷ Trường hợp 4
                      else sibling[prev-x] ← next-x
                                                ▷ Trường hợp 4
                    BINOMIAL-LINK(x, next-x)
                                                ▷ Trường hợp 4
                    x ← next-x                  ▷ Trường hợp 4
       next-x ← sibling[x]
return H
```

Hình 20.5 nêu một ví dụ về BINOMIAL-HEAP-UNION ở đó cả bốn trường hợp đã cho trong mã giả đều xảy ra.

Thủ tục BINOMIAL-HEAP-UNION có hai giai đoạn. Giai đoạn đầu, được thực hiện bởi lệnh gọi BINOMIAL-HEAP-MERGE, trộn các danh sách gốc của các đống nhị thức H_1 và H_2 thành một danh sách nối kết đơn lẻ H được sắp xếp theo độ theo thứ tự tăng đơn điệu. Tuy nhiên, có thể có tới hai gốc (nhưng không hơn) cho mỗi độ, do đó giai đoạn thứ hai nối kết các gốc có độ bằng nhau cho đến khi còn lại tối đa một gốc cho mỗi độ. Bởi vì danh sách nối kết H được sắp xếp theo độ, nên ta có thể nhanh chóng thực hiện tất cả các phép toán nối kết.

Về chi tiết, thủ tục làm việc như sau. Các dòng 1-3 bắt đầu bằng tiến trình trộn các danh sách gốc của các đống nhị thức H_1 và H_2 thành một danh sách gốc đơn lẻ H. Các danh sách gốc của H_1 và H_2 được sắp xếp theo độ tăng ngặt, và BINOMIAL-HEAP-MERGE trả về một danh sách gốc H được sắp xếp theo độ tăng đơn điệu. Nếu các danh sách gốc của H_1 và H_2 có m gốc chung với nhau, BINOMIAL-HEAP-MERGE chạy trong $O(m)$ thời gian bằng cách liên tục xét các gốc tại các đầu của hai danh sách gốc và chấp gốc có độ thấp hơn với danh sách gốc kết xuất, gỡ bỏ nó ra khỏi danh sách gốc nhập liệu của nó trong tiến trình.

Sau đó, thủ tục BINOMIAL-HEAP-UNION khởi tạo vài biến trỏ vào danh sách gốc của H. Trước tiên, nó đơn giản trở về trong các dòng 4-5 nếu như tình cờ hợp nhất hai đống nhị thức trống. Do đó, từ dòng 6 trở đi, ta biết H ít nhất sẽ có một gốc. Suốt thủ tục, ta duy trì ba biến trỏ vào danh sách gốc:

- x trỏ đến gốc hiện được xem xét,
- *prev-x* trỏ đến gốc đứng trước x trên danh sách gốc: *sibling*[*prev-x*] = x,

Trường hợp 3

và

Trường hợp 2

- *next-x* trỏ đến gốc theo sau x trên danh sách gốc: *sibling*[x] = *next-x*.

Thoạt đầu, ta có tối đa hai gốc trên danh sách gốc H của một độ đã cho: bởi vì H_1 và H_2 là các đống nhị thức, nên chúng chỉ có một gốc có một độ đã cho. Hơn nữa, BINOMIAL-HEAP-MERGE bảo đảm nếu hai gốc trong H có cùng độ, chúng sẽ kề nhau trong danh sách gốc.

Thực vậy, trong khi thi hành BINOMIAL-HEAP-UNION, có thể có ba gốc của một độ đã cho xuất hiện trên danh sách gốc H tại một thời gian nào đó. Dưới đây, ta sẽ thấy tình huống này có thể xảy ra như thế nào. Do đó, tại mỗi lần lặp lại của vòng lặp **while** trong các dòng 9-21,

ta quyết định xem có nên nối kết x và *next-x* hay không dựa trên các độ của chúng và có thể dựa trên độ của *sibling*[*next-x*]. Một bất biến của vòng lặp đó là mỗi lần ta bắt đầu thân của vòng lặp, cả x lẫn *next-x* đều không NIL.

Trường hợp 1, nêu trong Hình 20.6(a), xảy ra khi *degree*[x] ≠ *degree*[*next-x*], nghĩa là, khi x là gốc của một cây B_k và *next-x* là gốc của một cây B_l với một $l > k$. Các dòng 11-12 điều quản trường hợp này. Ta không nối kết x và *next-x*, do đó ta đơn giản dẫn các biến trỏ đi thêm một vị trí đổ xuống danh sách. Dòng 21 sẽ điều quản tiến trình cập nhật *next-x* để trỏ đến nút theo sau nút mới x, đây là chung cho mọi trường hợp.

Trường hợp 2, nêu trong Hình 20.6(b), xảy ra khi x là gốc đầu tiên trong ba gốc có độ bằng nhau, nghĩa là, khi

degree[x] = *degree*[*next-x*] = *degree*[*sibling*[*next-x*]] .

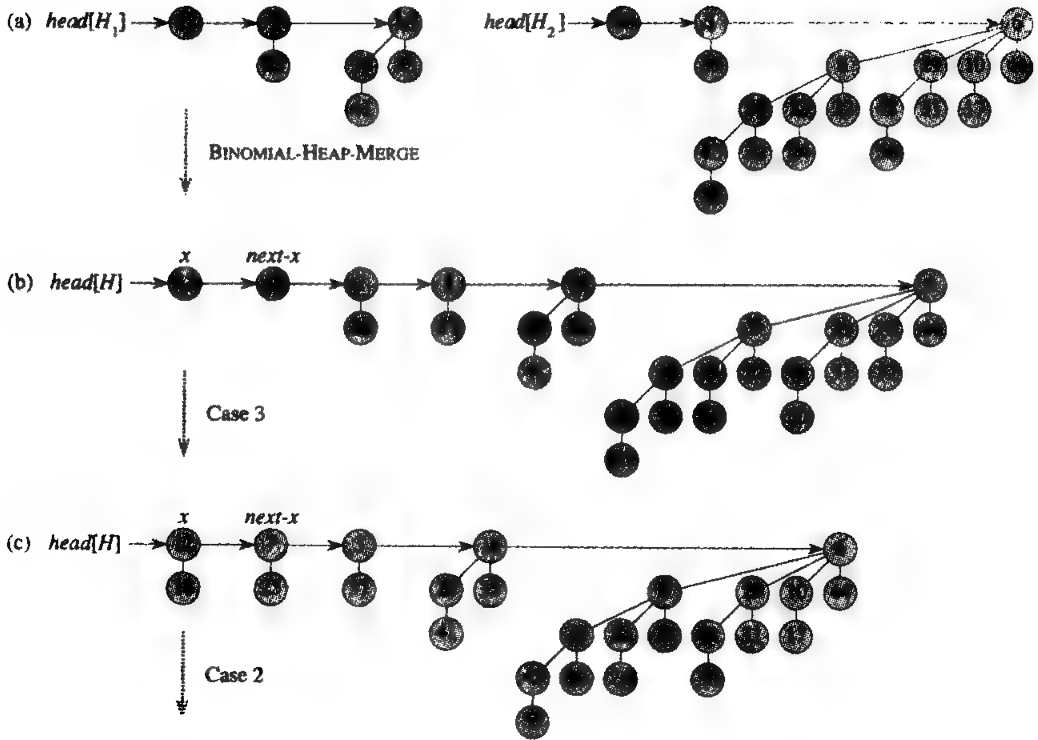
Ta điều quản trường hợp này giống như trường hợp 1: chỉ việc dẫn các biến trỏ đi thêm một vị trí đổ xuống danh sách. Dòng 10 kiểm tra cả hai trường hợp 1 và 2, và các dòng 11-12 điều quản cả hai trường hợp.

Các trường hợp 3 và 4 xảy ra khi x là gốc đầu tiên trong hai gốc có độ bằng nhau, nghĩa là, khi

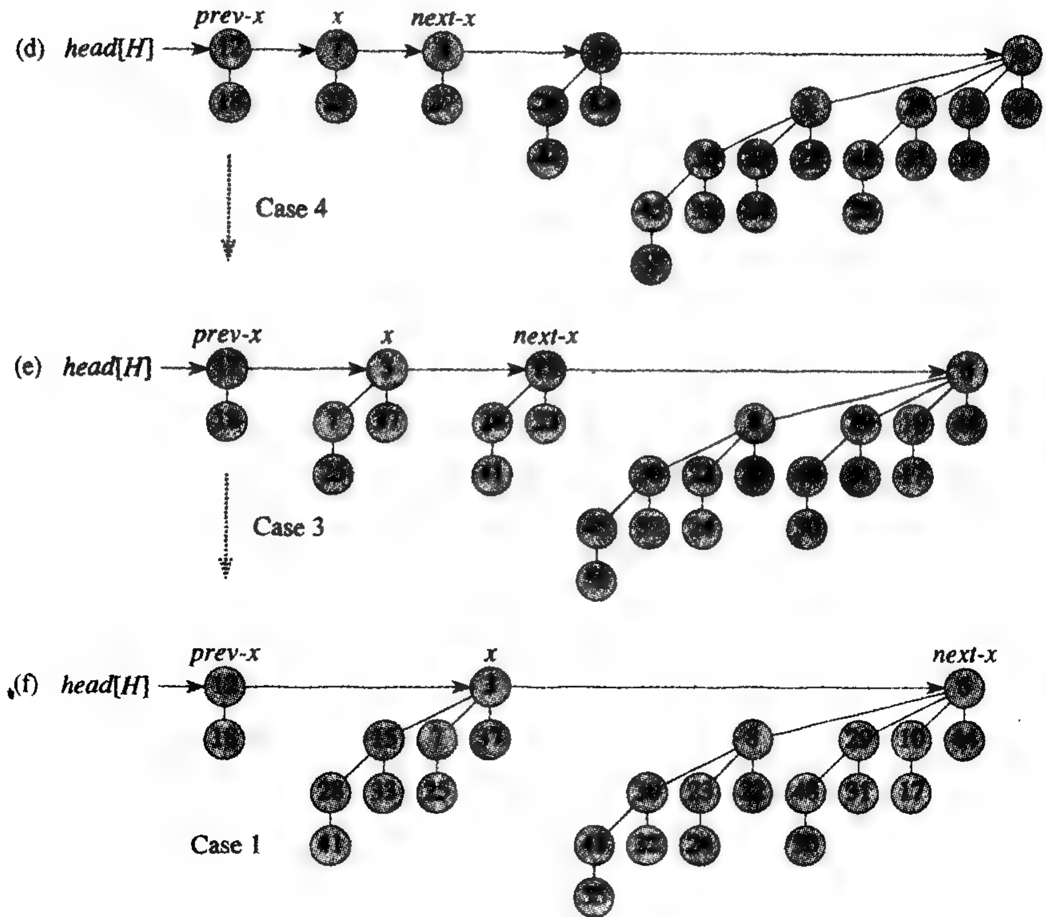
degree[x] = *degree*[*next-x*] ≠ *degree*[*sibling*[*next-x*]] .

Các trường hợp này có thể xảy ra trên lần lặp lại kế tiếp theo bất kỳ trường hợp nào, nhưng một trong số chúng luôn xảy ra ngay sau trường hợp 2. Trong các trường hợp 3 và 4, ta nối kết x và *next-x*. Hai trường hợp được phân biệt bởi x hoặc *next-x* có khóa nhỏ hơn, xác định nút sẽ là gốc sau khi cả hai nối kết.

Trong trường hợp 3, nêu trong Hình 20.6(c), *key*[x] ≤ *key*[*next-x*], do đó *next-x* được nối kết với x. Dòng 14 gỡ bỏ *next-x* ra khỏi danh sách gốc, và dòng 15 khiến *next-x* trở thành con mút trái của x.



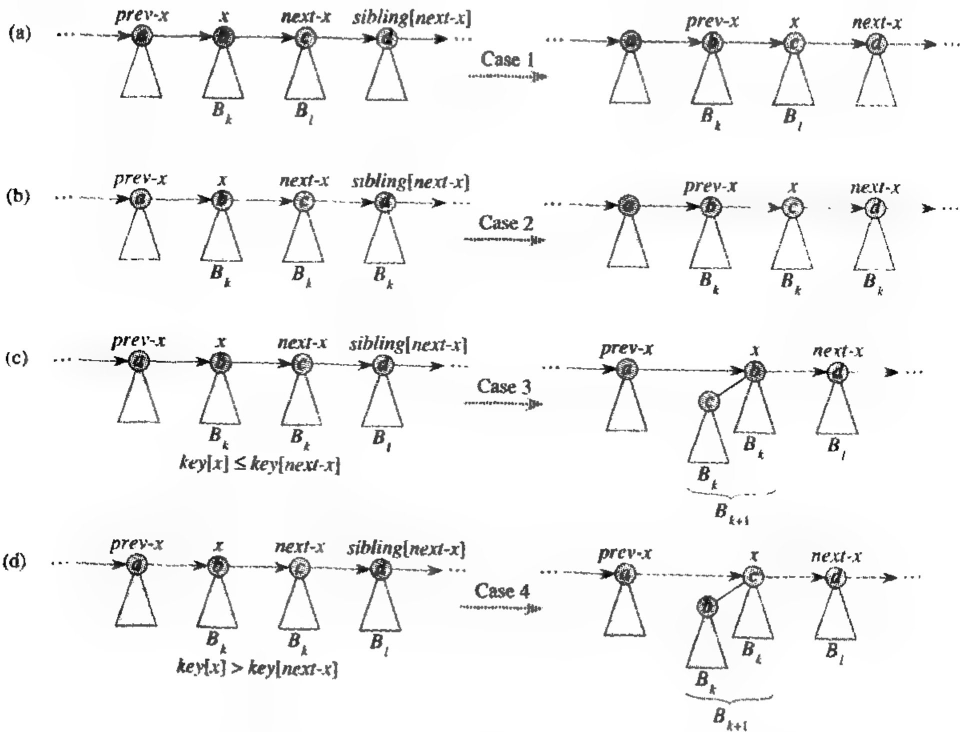
Hình 20.5 Tiến trình thi hành BINOMIAL-HEAP-UNION. **(a)** Các đống nhị thức H_1 và H_2. **(b)** Đống nhị thức H là kết xuất của BINOMIAL-HEAP-MERGE(H_1, H_2). Thoạt đầu, x là gốc đầu tiên trên danh sách gốc của H. Do cả x lẫn *next-x* đều có độ 0 và *key*[*x*] < *key*[*next-x*], nên trường hợp 3 được áp dụng. **(c)** Sau khi xảy ra nối kết, x là gốc đầu tiên trong ba gốc có cùng độ, do đó trường hợp 2 được áp dụng. **(d)** Sau khi tất cả các biến trỏ dời xuống một vị trí trong danh sách gốc, trường hợp 4 được áp dụng, bởi x là gốc đầu tiên của hai gốc có độ bằng nhau. **(e)** Sau khi nối kết xảy ra, trường hợp 3 được áp dụng. **(f)** Sau một nối kết khác, trường hợp 1 được áp dụng, bởi x có độ 3 và *next-x* có độ 4. Lần lặp lại này của vòng lặp **while** là chót, bởi sau khi các biến trỏ dời xuống một vị trí trong danh sách gốc, *next-x* = NIL.



Trong trường hợp 4, nêu trong Hình 20.6(d), *next-x* có khóa nhỏ hơn, do đó *x* được nối kết với *next-x*. Các dòng 16-18 gỡ bỏ *x* ra khỏi danh sách gốc, có hai trường hợp tùy thuộc vào việc *x* là gốc đầu tiên trên danh sách (dòng 17) hay không phải (dòng 18). Sau đó, dòng 19 khiến *x* trở thành con mút trái của *next-x*, và dòng 20 cập nhật *x* cho lần lặp lại kế tiếp.

Theo trường hợp 3 hoặc trường hợp 4, việc xác lập cho lần lặp lại kế tiếp của vòng lặp **while** là giống nhau. Ta vừa nối kết hai cây B_k- để tạo thành một cây B_{k+1}, mà giờ đây *x* trỏ đến nó. Đã có zero, một, hoặc hai cây B_{k+1} khác trên danh sách gốc từ kết xuất của BINOMIAL-HEAP-MERGE, do đó giờ đây *x* là gốc đầu tiên của một, hai, hoặc ba cây B_{k+1} trên danh sách gốc. Nếu *x* là gốc duy nhất, thì ta nhập trường hợp 1 trong lần lặp lại kế tiếp: $degree[x] \neq degree[next\text{-}x]$. Nếu *x* là gốc đầu tiên trong hai, thì ta nhập trường hợp 3 hoặc trường hợp 4 trong lần lặp lại kế tiếp. Nếu *x* là gốc đầu tiên trong ba, thì chính là lúc ta nhập trường hợp 2 trong lần lặp lại kế tiếp.

Thời gian thực hiện của BINOMIAL-HEAP-UNION là $O(\lg n)$, ở đó n là tổng các nút trong các đống nhị thức H_1 và H_2. Ta có thể xem nó như sau. Cho H_1 chứa n_1 các nút và H_2 chứa n_2 nút, sao cho $n = n_1 + n_2$. Như vậy, H_1 chứa tối đa $\lfloor \lg n_1 \rfloor + 1$ gốc và H_2 chứa tối đa $\lfloor \lg n_2 \rfloor + 1$ gốc, do đó H chứa tối đa $\lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor + 2 \leq 2\lfloor \lg n \rfloor + 2 = O(\lg n)$ gốc ngay sau lệnh gọi BINOMIAL-HEAP-MERGE. Như vậy, thời gian thực hiện BINOMIAL-HEAP-MERGE là $O(\lg n)$. Mỗi lần lặp lại của vòng lặp **while** chiếm $O(1)$ thời gian, và có tối đa $\lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor + 2$ lần lặp lại bởi mỗi lần lặp lại đẩy các biến trỏ tới một vị trí đổ xuống danh sách gốc của H hoặc gỡ bỏ một gốc ra khỏi danh sách gốc. Như vậy, tổng thời gian là $O(\lg n)$.



Hình 20.6 Bốn trường hợp xảy ra trong BINOMIAL-HEAP-UNION. Các nhãn a, b, c, và d chỉ được dùng để định danh các gốc có liên quan: chúng không nêu rõ các độ hoặc các khóa của các gốc này. Trong mỗi trường hợp, x là gốc của một cây B_k và $l > k$. **(a)** Trường hợp 1: $degree[x] \neq degree[next\text{-}x]$. Các biến trỏ dời tới một vị trí đổ xuống danh sách gốc. **(b)** Trường hợp 2: $degree[x] = degree[next\text{-}x] = degree[sibling[next\text{-}x]]$. Một lần nữa, các biến trỏ dời tới một vị trí đổ xuống danh sách, và lần lặp lại kế tiếp thi hành trường hợp 3 hoặc trường hợp 4. **(c)** Trường hợp 3: $degree[x] = degree[next\text{-}x] \neq degree[sibling[next\text{-}x]]$ và $key[x] \leq key[next\text{-}x]$. Ta gỡ bỏ *next-x* ra khỏi danh sách gốc và nối kết nó với x, tạo một cây B_{k+1}. **(d)** Trường hợp 4: $degree[x] = degree[next\text{-}x] \neq degree[sibling[next\text{-}x]]$ và $key[next\text{-}x] \leq key[x]$. Ta gỡ bỏ x ra khỏi danh sách gốc và nối kết nó với *next-x*, một lần nữa tạo ra một cây B_{k+1}.

Chèn một nút

Thủ tục dưới đây chèn nút x vào đống nhị thức H, tất nhiên ta mặc nhận nút x đã được phân bổ và $key[x]$ đã được điền.

```
BINOMIAL-HEAP-INSERT(H, x)
H' ← MAKE-BINOMIAL-HEAP()
p[x] ← NIL
child [x] ← NIL
sibling[x] ← NIL
degree[x] ← 0
head[H'] ← x
H - BINOMIAL-HEAP-UNION(H, H')
```

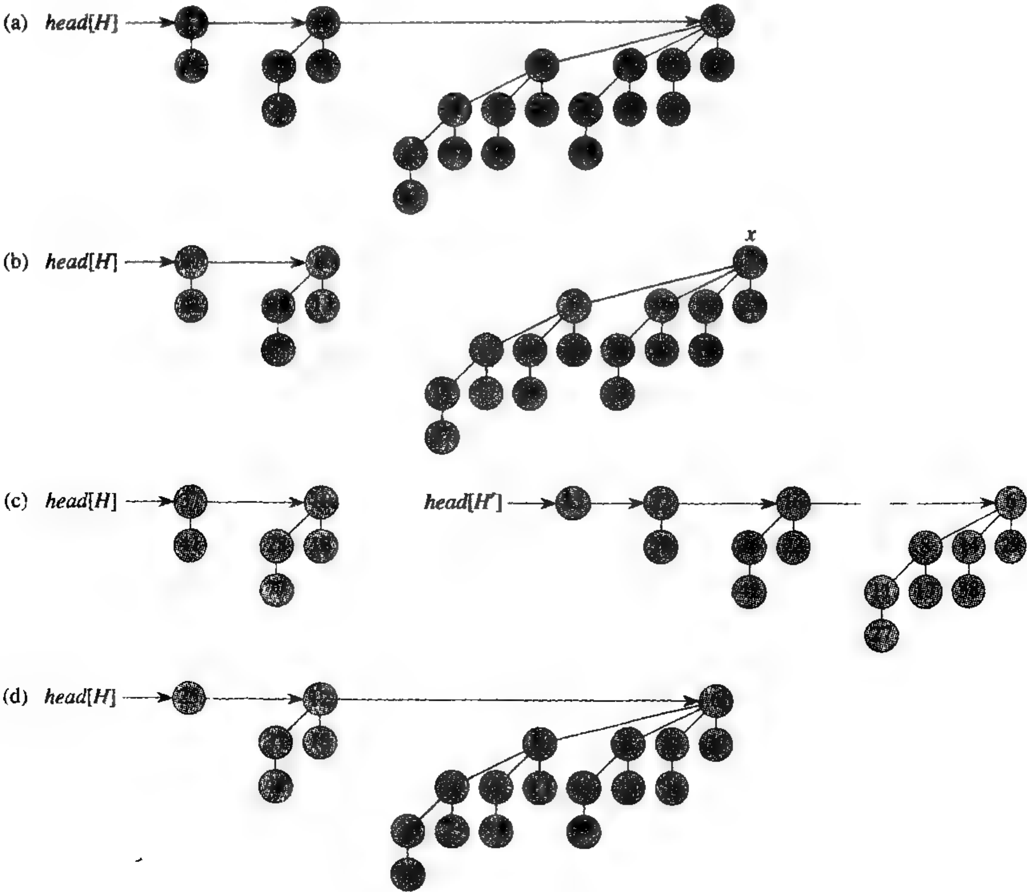
Thủ tục đơn giản tạo một đống nhị thức một nút H' trong $O(1)$ thời gian và hợp nhất nó với đống nhị thức H n-nút trong $O(\lg n)$ thời gian. Lệnh gọi BINOMIAL-HEAP-UNION sẽ đảm trách việc giải phóng đống nhị thức tạm thời H'. (Ta để dành kiểu thực thi trực tiếp không gọi BINOMIAL-HEAP-UNION làm Bài tập 20.2-8.)

Trích nút có khóa cực tiểu

Thủ tục dưới đây trích nút có khóa cực tiểu từ đống nhị thức H và trả về một biến trỏ đến nút đã trích.

```
BINOMIAL-HEAP-EXTRACT-MIN(H)
1 tìm gốc x có khóa cực tiểu trong danh sách gốc của H,
        và gỡ bỏ x ra khỏi danh sách gốc của H
2 H' ← MAKE-BINOMIAL-HEAP()
3 nghịch đảo thứ tự danh sách nối kết các con của x,
        và ấn định head[H'] để trỏ đến đầu danh sách kết quả
4 H ← BINOMIAL-HEAP-UNION(H, H')
5 return x
```

Thủ tục này làm việc như đã nêu trong Hình 20.7. Đống nhị thức nhập liệu H được nêu trong Hình 20.7(a). Hình 20.7(b) nêu tình huống sau dòng 1: gốc x có khóa cực tiểu đã được gỡ bỏ ra khỏi danh sách gốc của H. Nếu x là gốc của một cây B_k, thì với tính chất 4 của Bổ đề 20.1, các con của x, từ trái qua phải, là các gốc của các cây B_{k-1}, B_{k-2},..., B_0. Hình 20.7(c) chứng tỏ bằng cách đảo ngược danh sách các con của x trong dòng 3, ta có một đống nhị thức H' chứa mọi nút trong cây của x ngoại trừ chính x. Bởi cây của x được gỡ bỏ ra khỏi H trong dòng 1, nên đống nhị thức là kết quả của việc hợp nhất H và H' trong dòng 4, nêu



Hình 20.7 Hành động của BINOMIAL-HEAP-EXTRACT-MIN. **(a)** Một đống nhị thức *H*. **(b)** Gốc *x* có khóa cực tiểu được gỡ bỏ ra khỏi danh sách gốc của *H*. **(c)** Danh sách nối kết các con của *x* được đảo ngược, cho ta một đống nhị thức *H'* khác. **(d)** Kết quả của tiến trình hợp nhất *H* và *H'*.

trong Hình 20.7(d), sẽ chứa tất cả các nút ban đầu trong H ngoại trừ x. Cuối cùng, dòng 5 trả về x.

Do mỗi dòng 1-4 mất $O(\lg n)$ thời gian nếu H có n nút, nên BINOMIAL-HEAP-EXTRACT-MIN chạy trong $O(\lg n)$ thời gian.

Giảm một khóa

Thủ tục dưới đây giảm khóa của một nút x trong một đống nhị thức H theo một giá trị k mới. Nó phát ra một lỗi nếu k lớn hơn khóa hiện hành của x.

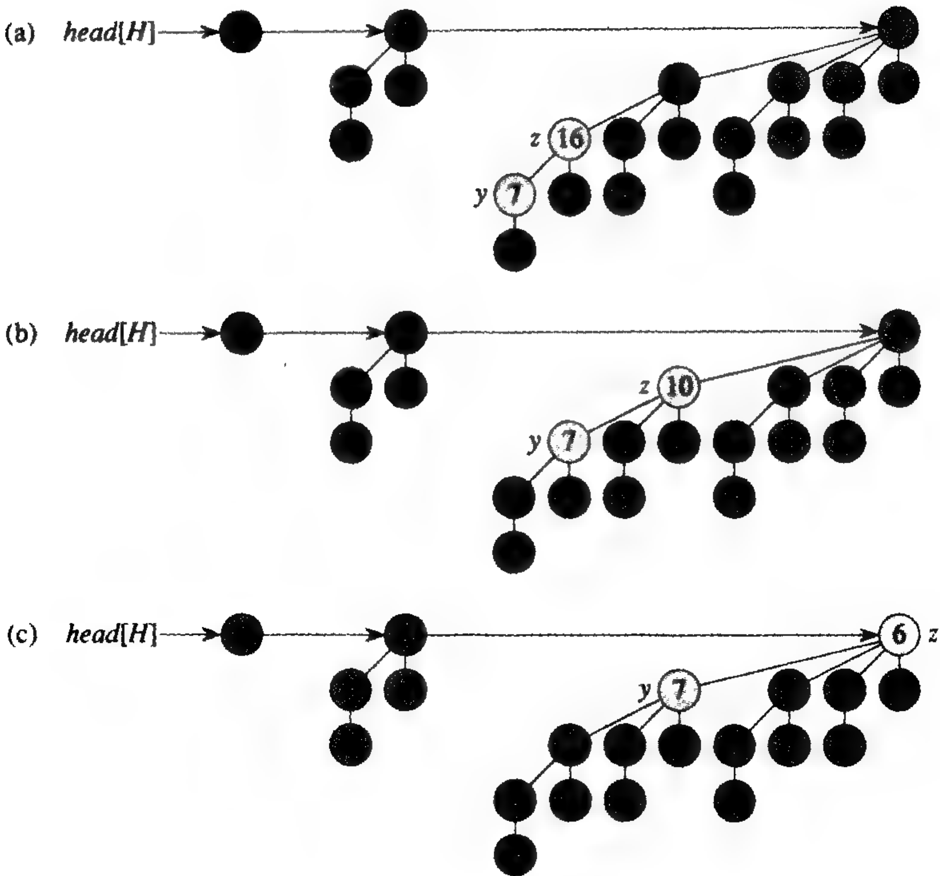
```
BINOMIAL-HEAP-DECREASE-KEY(H, x, k)
1   if k > key[x]
2       then error "khóa mới lớn hơn khóa hiện hành"
3   key[x] ← k
4   y ← x
5   z ← p[y]
6   while z ≠ NIL và key[y] < key[z]
7       do trao đổi key[y] ↔ key[z]
8           ▷ Nếu y và z có các trường vệ tinh, cũng trao đổi chúng.
9           y ← z
10          z ← p[y]
```

Như đã nêu trong Hình 20.8, thủ tục này giảm một khóa giống như trong một đống nhị phân: bằng cách "trồi bong bóng" khóa trong đống. Sau khi bảo đảm khóa mới thực tế không lớn hơn khóa hiện hành rồi gán khóa mới cho x, thủ tục tiến lên cây, với y thoạt tiên trỏ đến nút x. Trong mỗi lần lặp lại của vòng lặp **while** tại các dòng 6-10, $key[y]$ được kiểm tra dựa vào khóa của cha z của y. Nếu y là gốc hoặc $key[y] \geq key[z]$, cây nhị thức giờ đây được sắp xếp theo đống. Bằng không, nút y vi phạm tính năng sắp xếp theo đống, do đó khóa của nó được trao đổi với khóa của cha z của nó, cùng với mọi thông tin vệ tinh khác. Sau đó, thủ tục ấn định y theo z, tiến lên một cấp trong cây, và tiếp tục với lần lặp lại kế tiếp.

Thủ tục BINOMIAL-HEAP-DECREASE-KEY mất $O(\lg n)$ thời gian. Với tính chất 2 của Bổ đề 20.1, độ sâu cực đại của x là $\lfloor \lg n \rfloor$, do đó vòng lặp **while** của các dòng 6-10 sẽ lặp lại tối đa $\lfloor \lg n \rfloor$ lần.

Xóa một khóa

Ta có thể dễ dàng xóa khóa của một nút x và thông tin vệ tinh ra khỏi đống nhị thức H trong $O(\lg n)$ thời gian. Kiểu thực thi dưới đây mặc nhận không có nút nào hiện ở trong đống nhị thức có một khóa của $-\infty$.



Hình 20.8 Hành động của BINOMIAL-HEAP-DECREASE-KEY. **(a)** Tình huống ngay trước dòng 5 của lần lặp lại đầu tiên của vòng lặp **white**. Khóa của nút y đã giảm xuống 7, nhỏ hơn khóa của cha z của y. **(b)** Các khóa của hai nút được trao đổi, và tình huống ngay trước dòng 5 của lần lặp lại thứ hai được nêu. Các biến trỏ y và z đã dời lên một cấp trong cây, nhưng thứ tự đống vẫn bị vi phạm. **(c)** Sau một trao đổi khác và dời các biến trỏ y và z lên thêm một cấp, cuối cùng ta thấy rằng thứ tự đống được thỏa, do đó vòng lặp **while** kết thúc.

BINOMIAL-HEAP-DELETE(H, x)

1 BINOMIAL-HEAP-DECREASE-KEY($H, x, -\infty$)

2 BINOMIAL-HEAP-EXTRACT-MIN(H)

Thủ tục BINOMIAL-HEAP-DELETE tạo nút x có khóa cực tiểu duy nhất trong nguyên cả đống nhị thức bằng cách gán cho nó một khóa của $-\infty$. (Bài tập 20.2-6 xử lý tình huống ở đó $-\infty$ không thể xuất hiện dưới dạng một khóa, thậm chí tạm thời.) Sau đó, nó sủi bong bóng khóa này và thông tin vệ tinh kết hợp lên đến một gốc bằng cách gọi BINO-

MIAL-HEAP-DECREASE-KEY. Sau đó, gốc này được gỡ bỏ ra khỏi *H* bằng một lệnh gọi BINOMIAL-HEAP-EXTRACT-MIN. Thủ tục BINOMIAL-HEAP-DELETE mất $O(\lg n)$ thời gian.

Bài tập

20.2-1

Nêu một ví dụ về hai đống *nhị phân* có *n* thành phần cho mỗi đống sao cho BUILD-HEAP mất $\Theta(n)$ thời gian trên phép ghép nối các mảng của chúng.

20.2-2

Viết mã giả cho BINOMIAL-HEAP-MERGE.

20.2-3

Nêu đống nhị thức là kết quả khi một nút có khóa 24 được chèn vào đống nhị thức nêu trong Hình 20.7(d).

20.2-4

Nêu đống nhị thức là kết quả khi nút có khóa 28 được xóa ra khỏi đống nhị thức nêu trong Hình 20.8(c).

20.2-5

Giải thích tại sao thủ tục BINOMIAL-HEAP-MINIMUM không thể làm việc đúng đắn nếu các khóa có thể có giá trị ∞. Viết lại mã giả để nó làm việc đúng đắn trong các trường hợp như vậy.

20.2-6

Giả sử không có cách nào để biểu diễn khóa $-\infty$. Viết lại thủ tục BINOMIAL-HEAP-DELETE để làm việc đúng đắn trong tình huống này. Nó vẫn mất $O(\lg n)$ thời gian.

20.2-7

Tranh luận mối quan hệ giữa việc chèn vào một đống nhị thức và gia số một số nhị phân và mối quan hệ giữa việc hợp nhất hai đống nhị thức và cộng hai số nhị phân.

20.2-8

Căn cứ vào Bài tập 20.2-7, viết lại BINOMIAL-HEAP-INSERT để chèn một nút trực tiếp vào một đống nhị thức mà không gọi BINOMIAL-HEAP-UNION.

20.2-9

Chứng tỏ nếu các danh sách gốc được duy trì theo thứ tự giảm ngặt theo độ (thay vì thứ tự tăng ngặt), từng phép toán đống nhị thức có thể được thực thi mà không phải thay đổi thời gian thực hiện tiệm cận của nó.

20.2-10

Tìm các nhập liệu khiến BINOMIAL-HEAP-EXTRACT-MIN, BINOMIAL-HEAP-DECREASE-KEY, và BINOMIAL-HEAP-DELETE chạy trong $\Omega(\lg n)$ thời gian. Giải thích tại sao thời gian thực hiện ca xấu nhất của BINOMIAL-HEAP-INSERT, BINOMIAL-HEAP-MINIMUM, và BINOMIAL-HEAP-UNION là $\overset{\infty}{\Omega}(\lg n)$ mà không phải là $\Omega(\lg n)$. (Xem Bài toán 2-5.)

Các Bài Toán

20-1 Các đống 2-3-4

Chương 19 đã giới thiệu cây 2-3-4, ở đó mọi nút trong (ngoài gốc khả dĩ) có hai, ba, hoặc bốn con và tất cả các lá có cùng độ sâu. Trong bài toán này, ta sẽ thực thi các đống 2-3-4, hỗ trợ các phép toán đống khả trộn.

Các đống 2-3-4 khác với các cây 2-3-4 theo các cách sau đây. Trong các đống 2-3-4, chỉ các lá lưu trữ các khóa, và mỗi lá x lưu trữ chính xác một khóa trong trường *key*[x]. Không có cách sắp xếp thứ tự cụ thể nào đối với các khóa trong các lá; nghĩa là, từ trái qua phải, các khóa có thể theo một thứ tự bất kỳ. Mỗi nút trong x chứa một giá trị *small* [x] bằng với khóa nhỏ nhất được lưu trữ trong một lá bất kỳ trong cây con có gốc tại x. Gốc r chứa một trường *height*[r] là chiều cao của cây. Cuối cùng, chủ trương của các đống 2-3-4 là lưu giữ trong bộ nhớ chính, sao cho không cần các lần đọc và ghi đĩa.

Thực thi các phép toán đống 2-3-4 dưới đây. Mỗi phép toán trong các phần (a)-(e) sẽ chạy trong $O(\lg n)$ thời gian trên một đống 2-3-4 có n thành phần. Phép toán UNION trong phần (f) sẽ chạy trong $O(\lg n)$ thời gian, ở đó n là số lượng các thành phần trong hai đống nhập liệu.

a. MINIMUM, trả về một biến trỏ đến lá có khóa nhỏ nhất.

b. DECREASE-KEY, giảm khóa của một lá x đã cho theo một giá trị $k \le key[x]$ đã cho.

c. INSERT, chèn lá x có khóa k.

d. DELETE, xóa một lá x đã cho.

e. EXTRACT-MIN, trích lá có khóa nhỏ nhất.

f. UNION, hợp nhất hai đống 2-3-4, trả về một đống 2-3-4 đơn lẻ và hủy các đống nhập liệu.

20-2 Thuật toán cây tỏa nhánh cực tiểu dùng các đống khả trộn

Chương 24 trình bày hai thuật toán để giải quyết bài toán tìm một cây tỏa nhánh cực tiểu của một đồ thị không có hướng. Ở đây, ta sẽ xem cách dùng các đống khả trộn để nghĩ ra một thuật toán cây tỏa nhánh cực tiểu khác.

Ta có một đồ thị liên thông không có hướng $G = (V, E)$ có một hàm trọng số $w. E \rightarrow \mathbf{R}$. Ta gọi $w(u, v)$ là trọng số của cạnh (u, v). Ta muốn tìm một cây tỏa nhánh cực tiểu với G: một tập hợp con phi chu trình $T \subseteq E$ liên thông tất cả các đỉnh trong V có tổng trọng số

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

được giảm thiểu.

Mã giả dưới đây, có thể được chứng minh là đúng bằng các kỹ thuật trong Đoạn 24.1, sẽ liên tục một cây tỏa nhánh cực tiểu T. Nó duy trì một phân hoạch $\{V_i\}$ các đỉnh của V và, với mỗi tập hợp V_i, một tập hợp

$E_i \subseteq \{(u,v): u \in V$ hoặc $v \in V_i\}$

gồm các cạnh liên thuộc trên các đỉnh trong V_i.

```
MST-MERGEABLE-HEAP(G)
T ← ∅
for mỗi đỉnh v_i ∈ v[G]
      do V_i ← {v_i}
         E_i ← {(v_i,v) ∈ E[G]}
while có trên một tập hợp V_i
      do chọn bất kỳ tập hợp V_i
         trích cạnh trọng số cực tiểu (u, v) từ E_i
         mặc nhận mà không mất tính tổng quát u ∈ V_i và v ∈ V_j
         if i ≠ j
            then T ← T ∪ {(u, v)}
                 V_i ← V_i ∪ V_j, hủy V_j
```

$$E_i \leftarrow E_i \cup E_j$$

Mô tả cách thực thi thuật toán này bằng các phép toán đống khả trộn đã cho trong Hình 20.1. Nêu thời gian thực hiện của kiểu thực thi của bạn, giả định các đống khả trộn được thực thi bằng các đống nhị thức.

Ghi chú Chương

Các đống nhị thức đã được Vuillemin [196] giới thiệu vào năm 1978. Brown [36, 37] đã nghiên cứu chi tiết các tính chất của chúng.

21 Các Đống Fibonacci

Trong Chương 20, ta đã tìm hiểu cách thức mà các đống nhị thức hỗ trợ, trong $O(\lg n)$ thời gian trường hợp xấu nhất, các phép toán đống khả trộn INSERT, MINIMUM, EXTRACT-MIN, và UNION, cộng với các phép toán DECREASE-KEY và DELETE. Trong chương này, ta sẽ xét các đống Fibonacci, hỗ trợ các phép toán tương tự nhưng có ưu điểm đó là các phép toán không liên quan đến việc xóa một thành phần sẽ chạy trong $O(1)$ thời gian khấu trừ.

Theo quan điểm lý thuyết, các đống Fibonacci đặc biệt thỏa đáng khi số lượng phép toán EXTRACT-MIN và DELETE tương đối nhỏ so với số lượng các phép toán khác được thực hiện. Tình huống này nảy sinh trong nhiều ứng dụng. Ví dụ, một số thuật toán cho các bài toán đồ thị có thể gọi DECREASE-KEY một lần cho mỗi cạnh. Với các đồ thị dày đặc, có nhiều cạnh, thời gian khấu trừ $O(1)$ của mỗi lệnh gọi DECREASE-KEY sẽ bổ sung dần đến một mức cải thiện lớn so với $\Theta(\lg n)$ thời gian trường hợp xấu nhất của các đống nhị phân hoặc nhị thức. Cho đến nay, các thuật toán nhanh nhất theo tiệm cận cho các bài toán như tính toán các cây tỏa nhánh cực tiểu (Chương 24) hoặc tìm các lộ trình ngắn nhất nguồn đơn (Chương 25) chủ yếu vận dụng các đống Fibonacci.

Tuy nhiên, theo quan điểm thực tiễn, các thừa số bất biến và tính phức tạp về lập trình của các đống Fibonacci khiến chúng ít được ưa chuộng hơn so với các đống nhị phân bình thường (hoặc k-phân) cho hầu hết các ứng dụng. Như vậy, các đống Fibonacci chủ yếu được quan tâm về lý thuyết. Nếu phát triển một cấu trúc dữ liệu đơn giản hơn nhiều có cùng các cận thời gian khấu trừ như các đống Fibonacci, ắt nó sẽ được dùng nhiều trong thực tế.

Cũng như một đống nhị thức, một đống Fibonacci là một tập hợp các cây. Thực tế, các đống Fibonacci tự do dựa trên các đống nhị thức. Nếu cả DECREASE-KEY lẫn DELETE đều không được triệu gọi trên một đống Fibonacci, mỗi cây trong đống sẽ giống như một cây nhị thức. Tuy nhiên, các đống Fibonacci khác với các đống nhị thức ở chỗ chúng có một cấu trúc nới lỏng hơn, kể cả các biên thời gian tiệm cận đã cải tiến. Công trình duy trì cấu trúc đó có thể bị trì hoãn cho

đến khi thuận tiện để thực hiện.

Cũng như các bảng động của Đoạn 18.4, các đống Fibonacci cung cấp một ví dụ hay về một cấu trúc dữ liệu được thiết kế theo chủ trương phân tích khấu trừ. Trực giác và các cuộc phân tích các phép toán đống Fibonacci trong phần còn lại của chương này dựa nhiều vào phương pháp thế của Đoạn 18.3.

Phần giải thích trong chương này mặc nhận bạn đã đọc Chương 20 về các đống nhị thức. Các định chuẩn cho các phép toán xuất hiện trong chương đó, như bảng trong Hình 20.1, tóm tắt các cận thời gian cho các phép toán trên các đống nhị phân, các đống nhị thức, và các đống Fibonacci. Phần trình bày của chúng ta về cấu trúc của các đống Fibonacci dựa vào cấu trúc đống nhị thức. Bạn cũng sẽ thấy một số phép toán thực hiện trên các đống Fibonacci cũng tương tự như các phép toán thực hiện trên các đống nhị thức.

Cũng như các đống nhị thức, các đống Fibonacci không được thiết kế để hỗ trợ hiệu quả phép toán SEARCH; do đó các phép toán tham chiếu một nút đã cho cần phải có một biến trỏ đến nút đó như là một phần nhập liệu của chúng.

Đoạn 21.1 định nghĩa các đống Fibonacci, mô tả phần biểu diễn của chúng, và trình bày hàm thế được dùng cho kỹ thuật phân tích khấu trừ của chúng. Đoạn 21.2 nêu cách thực thi các phép toán đống khả trộn và cách đạt được các cận thời gian khấu trừ nêu trong Hình 20.1. Hai phép toán còn lại, DECREASE-KEY và DELETE, được trình bày trong Đoạn 21.3. Cuối cùng, Đoạn 21.4 hoàn tất phần chính của cuộc phân tích.

21.1 Cấu trúc của các đống Fibonacci

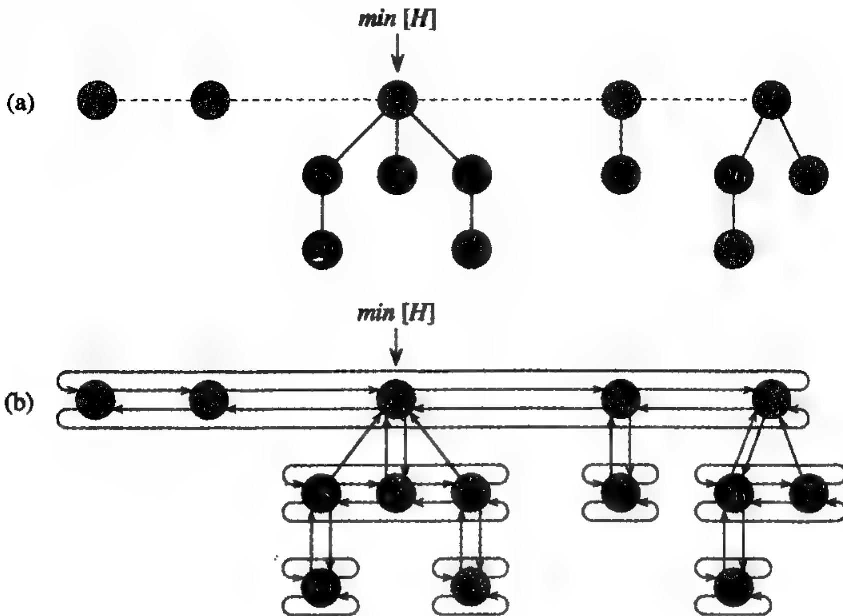
Cũng như một đống nhị thức, một ***đống Fibonacci*** là một tập hợp của các cây được sắp xếp theo đống. Tuy nhiên, các cây trong một đống Fibonacci không bị ràng buộc phải là các cây nhị thức. Hình 21.1(a) nêu một ví dụ về một đống Fibonacci.

Khác với các cây trong các đống nhị thức, có sắp xếp thứ tự, các cây trong các đống Fibonacci có gốc nhưng không được sắp xếp. Như Hình 21.1(b) đã nêu, mỗi nút x chứa một biến trỏ $p[x]$ đến cha của nó và một biến trỏ $child[x]$ đến một bất kỳ trong số các con của nó. Các con của x được nối kết với nhau theo một danh sách nối kết đôi theo vòng tròn, mà ta gọi là ***danh sách con*** của x. Mỗi con y trong một danh sách con có các biến trỏ $left[y]$ và $right[y]$ trỏ đến anh em ruột trái và phải của y, theo thứ tự nêu trên. Nếu nút y là một con duy nhất, thì $left[y] = right[y]$

= y. Thứ tự xuất hiện của các anh em ruột trong một danh sách con là tùy ý.

Các danh sách nối kết đôi theo vòng tròn (xem Đoạn 11.2) có hai ưu điểm để dùng trong các đống Fibonacci. Thứ nhất, ta có thể gỡ bỏ một nút ra khỏi một danh sách nối kết đôi theo vòng tròn trong $O(1)$ thời gian. Thứ hai, cho hai danh sách như vậy, ta có thể ghép nối chúng (hoặc "kết" chúng với nhau) thành một danh sách nối kết đôi theo vòng tròn trong $O(1)$ thời gian. Trong các phần mô tả về các phép toán đống Fibonacci, ta sẽ tham chiếu các phép toán này không chính thức, để người đọc điền các chi tiết của các cách thực thi của chúng.

Hai trường khác trong mỗi nút sẽ được đưa vào sử dụng. Số lượng các con trong danh sách con của nút x được lưu trữ trong *degree*[x]. Trường có giá trị bool *mark*[x] nêu rõ nút x đã mất một con hay chưa từ lần chót mà x đã được chuyển thành con của một nút khác. Ta sẽ không để tâm về các chi tiết đánh dấu các nút cho đến Đoạn 21.3. Các nút mới tạo không được đánh dấu, và một nút x trở thành không đánh dấu mỗi khi nó được chuyển thành con của một nút khác.



Hình 21.1 (a) Một đống Fibonacci bao gồm năm cây được sắp xếp theo đống và 14 nút. Vạch gạch cách nêu rõ danh sách gốc. Nút cực tiểu của đống là nút chứa khóa 3. Ba nút được đánh dấu là những nút được tô đen. Thế của đống Fibonacci cụ thể này là 5 + 2 • 3 = 11. **(b)** Một phần biểu diễn hoàn chỉnh hơn nêu các biến trỏ p (các mũi tên lên), *child* (các mũi tên xuống), *left* và *right* (các mũi tên chỉ sang hai phía). Các chi tiết này được bỏ qua trong các hình còn lại của chương này, bởi tất cả các thông tin nêu ở đây có thể được xác định từ nội dung xuất hiện trong phần **(a)**.

Một đống Fibonacci H đã cho được truy cập bởi một biến trỏ $min[H]$ đến gốc của cây chứa một khóa cực tiểu; nút này được gọi là ***nút cực tiểu*** [minimum node] của đống Fibonacci. Nếu một đống Fibonacci H trống rỗng, thì $min[H]$ = NIL.

Các gốc của tất cả các cây trong một đống Fibonacci được nối kết với nhau bằng các biến trỏ *left* và *right* của chúng vào một danh sách nối kết đôi theo vòng tròn có tên là ***danh sách gốc*** của đống Fibonacci. Như vậy, biến trỏ $min[H]$ trỏ đến nút trong danh sách gốc có khóa là cực tiểu. Thứ tự của các cây trong một danh sách gốc là tùy ý.

Ta dựa vào một thuộc tính khác của một đống Fibonacci H: số lượng các nút hiện nằm trong H được lưu giữ trong $n[H]$.

Hàm thế

Như đã nêu, ta sẽ dùng phương pháp thế của Đoạn 18.3 để phân tích khả năng thực hiện của các phép toán đống Fibonacci. Với một đống Fibonacci H đã cho, ta nêu rõ số lượng các cây trong danh sách gốc của H là $t(H)$ và số lượng các nút được đánh dấu trong H là $m(H)$. Sau đó, thế của đống Fibonacci H được định nghĩa bằng

$$\Phi(H) = t(H) + 2\, m(H)\,. \quad (21.1)$$

Ví dụ, thế của đống Fibonacci nêu trong Hình 21.1 là $5 + 2 \cdot 3 = 11$. Thế của một tập hợp các đống Fibonacci chính là tổng các thế của các đống Fibonacci cấu thành của nó. Ta sẽ mặc nhận một đơn vị của thế có thể thanh toán cho một lượng công việc bất biến, ở đó hằng đủ lớn để trang trải mức hao phí của bất kỳ trong số các mẩu thời gian bất biến cụ thể của công việc mà ta có thể gặp.

Ta mặc nhận rằng một ứng dụng đống Fibonacci bắt đầu với không có đống nào. Do đó, thế ban đầu là 0, và theo phương trình (21.1), thế là không âm tại tất cả các thời gian sau đó. Như vậy, từ phương trình (18.2), một cận trên trên tổng mức hao phí khấu trừ sẽ là một cận trên trên tổng mức hao phí thực tế cho dãy các phép toán.

Độ cực đại

Các phân tích khấu trừ mà ta sẽ thực hiện trong các đoạn còn lại của chương này mặc nhận rằng có một cận trên $D(n)$ đã biết trên độ cực đại của bất kỳ nút nào trong một đống Fibonacci n-nút. Bài tập 21.2-3 chứng tỏ lúc chỉ hỗ trợ các phép toán đống khả trộn, $D(n) = \lfloor \lg n \rfloor$. Trong Đoạn 21.3, ta sẽ chứng tỏ khi ta hỗ trợ DECREASE-KEY và DELETE, $D(n) = O(\lg n)$.

21.2 Các phép toán đống khả trộn

Trong đoạn này, ta mô tả và phân tích các phép toán đống khả trộn như đã thực thi cho các đống Fibonacci. Nếu chỉ các phép toán này—MAKE-HEAP, INSERT, MINIMUM, EXTRACT-MIN, và UNION—được hỗ trợ, mỗi đống Fibonacci đơn giản là một tập hợp các cây nhị thức "không sắp xếp." Một ***cây nhị thức không sắp xếp*** cũng giống như một cây nhị thức, và nó cũng được định nghĩa một cách đệ quy. Cây nhị thức không sắp xếp U_0 bao gồm một nút đơn lẻ, và một cây nhị thức không sắp xếp U_k bao gồm hai cây nhị thức không sắp xếp U_{k-1} mà gốc của một cây được chuyển thành một con *bất kỳ* của gốc của cây kia. Bổ đề 20.1, cung cấp các tính chất của các cây nhị thức, cũng áp dụng cho các cây nhị thức không sắp xếp, nhưng với biến thể dưới đây cho tính chất 4 (xem Bài tập 21.2-2):

4'. Với cây nhị thức không sắp xếp U_k, gốc có độ k, lớn hơn so với của bất kỳ nút nào khác. Các con của gốc là các gốc của các cây con U_0, U_1,..., U_{k-1}, theo một thứ tự nào đó.

Như vậy, nếu một đống Fibonacci n-nút là một tập hợp các cây nhị thức không sắp xếp, thì $D(n) = \lg n$.

Ý tưởng chính trong các phép toán đống khả trộn trên các đống Fibonacci đó là trì hoãn công việc miễn là khả dĩ. Có một giá phải trả cho khả năng thực hiện giữa các cách thực thi của nhiều phép toán khác nhau. Nếu số lượng các cây trong một đống Fibonacci không lớn, ta có thể nhanh chóng xác định nút cực tiểu mới trong một phép toán EXTRACT-MIN. Tuy nhiên, như đã thấy với các đống nhị thức trong Bài tập 20.2-10, ta thanh toán một giá để bảo đảm số lượng các cây là nhỏ: có thể mất tới $\Omega(\lg n)$ thời gian để chèn một nút vào một đống nhị thức hoặc để hợp nhất hai đống nhị thức. Như sẽ thấy, ta không gắng thông nhất các cây trong một đống Fibonacci khi chèn một nút mới hoặc hợp nhất hai đống. Ta dành sự thống nhất cho phép toán EXTRACT-MIN, là khi ta thực sự cần tìm nút cực tiểu mới.

Tạo một đống Fibonacci mới

Để tạo một đống Fibonacci trống, thủ tục MAKE-FIB-HEAP phân bổ và trả về đối tượng đống Fibonacci H, ở đó $n[H] = 0$ và $min[H]$ = NIL; không có cây nào trong H. Bởi $t(H) = 0$ và $m(H) = 0$, nên thế của đống Fibonacci trống là $\Phi(H) = 0$. Như vậy, mức hao phí khấu trừ của MAKE-FIB-HEAP bằng với mức hao phí thực tế $O(1)$ của nó.

Chèn một nút

Thủ tục dưới đây chèn nút *x* vào đống Fibonacci *H*, tất nhiên mặc nhận nút đã được phân bổ và *key*[*x*] đã được điền.

```
FIB-HEAP-INSERT(H, x)
1   degree[x] ← 0
2   p[x] ← NIL
3   child[x] ← NIL
4   left[x] ← x
5   right[x] ← x
6   mark[x] ← FALSE
7   ghép nối danh sách gốc chứa x với danh sách gốc H
8   if min[H] = NIL hoặc key[x] < key[min[H]]
9       then min[H] ← x
10  n[H] ← n[H] + 1
```

Sau khi các dòng 1-6 khởi tạo các trường có cấu trúc của nút *x*, khiến nó trở thành danh sách nối kết đôi theo vòng tròn của riêng nó, dòng 7 bổ sung *x* vào danh sách gốc của *H* trong *O*(1) thời gian thực tế. Như vậy, nút *x* trở thành một cây nút đơn được sắp xếp theo đống, và như vậy là một cây nhị thức không sắp xếp, trong đống Fibonacci. Nó không có các con và không được đánh dấu. Sau đó, các dòng 8-9 cập nhật biến trỏ đến nút cực tiểu của đống Fibonacci *H* nếu cần. Cuối cùng, dòng 10 gia số *n*[*H*] để phản ánh việc bổ sung nút mới. Hình 21.2 nêu một nút có khóa 21 được chèn vào đống Fibonacci của Hình 21.1.

Khác với thủ tục BINOMIAL-HEAP-INSERT, FIB-HEAP-INSERT không gắng thống nhất các cây trong đống Fibonacci. Nếu *k* phép toán FIB-HEAP-INSERT xảy ra liên tiếp, thì *k* cây nút đơn được bổ sung vào danh sách gốc.



Hình 21.2 Chèn một nút vào một đống Fibonacci. **(a)** Một đống Fibonacci *H*. **(b)** Đống Fibonacci *H* sau khi chèn nút có khóa 21. Nút trở thành cây riêng của nó được sắp xếp theo đống và sau đó được bổ sung vào danh sách gốc, trở thành anh em ruột trái của gốc.

Để xác định mức hao phí khấu trừ của FIB-HEAP-INSERT, cho H là đống Fibonacci nhập liệu và H' là đống Fibonacci kết quả. Thì, $t(H') = t(H) + 1$ và $m(H') = m(H)$, và sự gia tăng trong thế là

$$((t(H) + 1) + 2\,m(H)) - (t(H) + 2\,m(H)) = 1 .$$

Bởi mức hao phí thực tế là $O(1)$, nên mức hao phí khấu trừ là $O(1) + 1 = O(1)$.

Tìm nút cực tiểu

Nút cực tiểu của một đống Fibonacci H được biến trỏ $min[H]$ cung cấp, do đó ta có thể tìm nút cực tiểu trong $O(1)$ thời gian thực tế. Do thế của H không thay đổi, nên mức hao phí khấu trừ của phép toán này bằng với $O(1)$ mức hao phí thực tế của nó.

Hợp nhất hai đống Fibonacci

Thủ tục dưới đây hợp nhất các đống Fibonacci H_1 và H_2, hủy H_1 và H_2 trong tiến trình.

```
FIB-HEAP-UNION(H1, H2)
1 H ← MAKE-FIB-HEAP()
2 min[H] ← min[H1]
3 ghép nối danh sách gốc của H2 với danh sách gốc của H
4 if (min[H1] = NIL) hoặc (min[H2] ≠ NIL và min[H2] < min[H1])
5     then min[H] ← min[H2]
6 n[H] ← n[H1] + n[H2]
7 giải phóng các đối tượng H1 và H2
8 return H
```

Các dòng 1-3 ghép nối các danh sách gốc của H_1 và H_2 thành một danh sách gốc mới H. Các dòng 2, 4, và 5 ấn định nút cực tiểu của H, và dòng 6 ấn định $n[H]$theo tổng số các nút. Các đối tượng đống Fibonacci H_1 và H_2 được giải phóng trong dòng 7, và dòng 8 trả về đống Fibonacci kết quả H. Như trong thủ tục FIB-HEAP-INSERT, không xảy ra sự thống nhất các cây.

Thay đổi trong thế là

$$\begin{aligned}\Phi(H) \; - \; & (\Phi(H_1) + \Phi(H_2)) \\ = \; & (t(H) + 2\,m(H)) - ((t(H_1) + 2\,m(H_1)) + (t(H_2) + 2\,m(H_2))) \\ = \; & 0,\end{aligned}$$

bởi $t(H) = t(H_1)+t(H_2)$ và $m(H) = m(H_1)+m(H_2)$. Do đó, mức hao phí khấu trừ của FIB-HEAP-UNION bằng với mức hao phí thực tế $O(1)$ của nó.

Trích nút cực tiểu

Tiến trình trích nút cực tiểu là dạng phức tạp nhất của các phép toán được trình bày trong đoạn này. Nó cũng là nơi chung cuộc xảy ra công việc thông nhất các cây đã bị trì hoãn trong danh sách gốc. Mã giả dưới đây trích nút cực tiểu. Để tiện dụng, mã mặc nhận khi được gỡ bỏ một nút ra khỏi một danh sách nối kết, các biến trỏ còn lại trong danh sách sẽ được cập nhật, nhưng các biến trỏ trong nút đã trích sẽ được giữ nguyên không đổi. Nó cũng sử dụng thủ tục phụ CONSOLIDATE, sẽ được trình bày ngắn gọn.

```
FIB-HEAP-EXTRACT-MIN(H)
1   z ← min[H]
2   if z ≠ NIL
3     then với mỗi con x của z
4              do cộng x vào danh sách gốc của H
5                 p[x] ← NIL
6          gỡ bỏ z ra khỏi danh sách gốc của H
7          if z = right[z]
8              then min[H] ← NIL
9              else min[H] ← right[z]
10                 CONSOLIDATE(H)
11         n[H] ← n[H] - 1
12  return z
```

Như đã nêu trong Hình 21.3, FIB-HEAP-EXTRACT-MIN làm việc bằng cách trước tiên tạo một gốc từ mỗi trong số các con của nút cực tiểu và gỡ bỏ nút cực tiểu ra khỏi danh sách gốc. Sau đó, nó thống nhất danh sách gốc bằng cách nối kết các gốc có độ bằng nhau cho đến khi còn lại tối đa một gốc từ mỗi độ.

Ta bắt đầu tại dòng 1 bằng cách lưu một biến trỏ z vào nút cực tiểu; biến trỏ này được trả về tại cuối. Nếu z = NIL, thì đống Fibonacci H đã trống sẵn và ta đã hoàn tất. Bằng không, như trong thủ tục BINOMIAL-HEAP-EXTRACTMIN, ta xóa nút z ra khỏi H bằng cách khiến tất cả các con của z thành các gốc của H trong các dòng 3-5 (đặt chúng vào danh sách gốc) và gỡ bỏ z ra khỏi danh sách gốc trong dòng 6. Nếu z = *right*[z] sau dòng 6, thì z là nút duy nhất trên danh sách gốc và nó không có các con, do đó tất cả còn lại đó là biến đống Fibonacci trở thành trống trong dòng 8 trước khi trả về z. Bằng không, ta ấn định biến trỏ *min*[H] vào danh sách gốc để trỏ đến một nút khác ngoài z (trong trường

hợp này là *right*[z]). Hình 21.3(b) nêu đống Fibonacci của Hình 21.3(a) sau khi dòng 9 đã được thực hiện.

Bước kế tiếp, ở đó ta rút gọn số lượng các cây trong đống Fibonacci, đó là ***thống nhất*** danh sách gốc của *H*; điều này được thực hiện bằng lệnh gọi CONSOLIDATE(*H*). Tiến trình thống nhất danh sách gốc bao gồm việc liên tục thi hành các bước sau đây cho đến khi mọi gốc trong danh sách gốc có một giá trị *degree* riêng biệt.

1. Tìm hai gốc x và y trong danh sách gốc có cùng độ, ở đó $key[x] \leq key[y]$.

2. ***Nối kết*** y với x: gỡ bỏ y ra khỏi danh sách gốc, và chuyển y thành một con của x. Phép toán này được thực hiện bởi thủ tục FIB-HEAP-LINK. Trường *degree*[x] được gia số, và dấu trên y bị xóa, nếu có.

Thủ tục CONSOLIDATE sử dụng một mảng phụ $A[0..D(n[H])$; nếu $A[i] = y$, thì y hiện là một gốc có $degree[y] = i$.

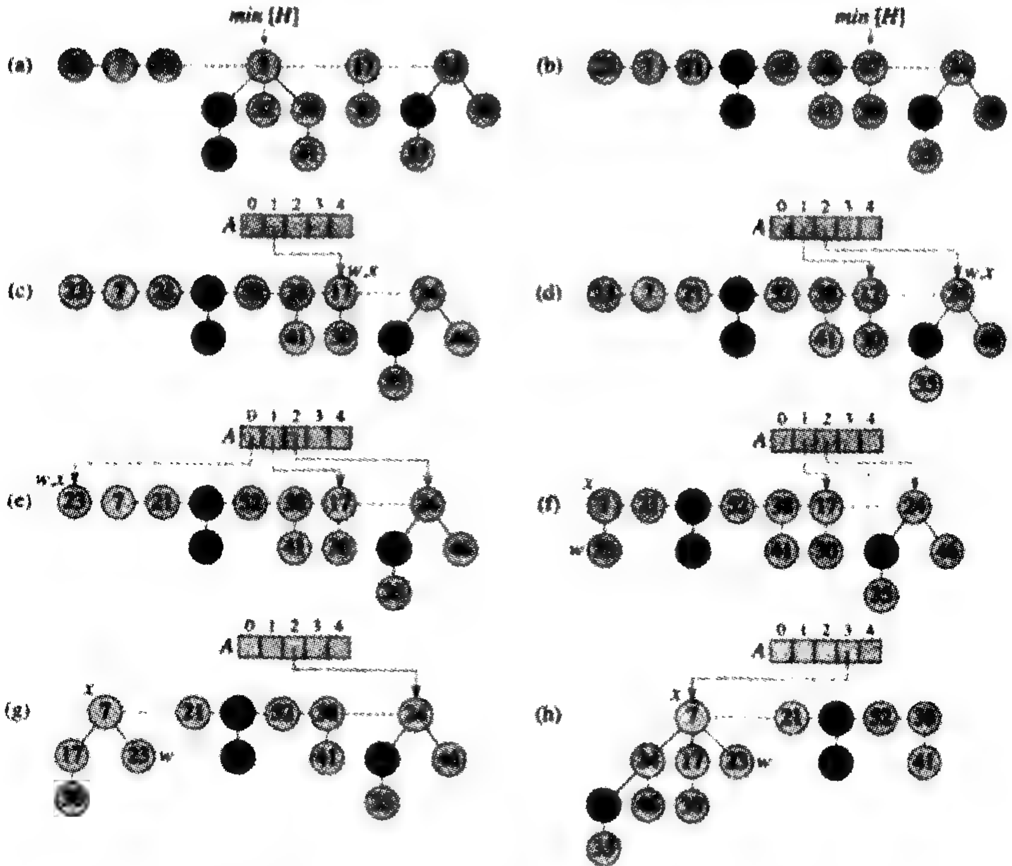
```
CONSOLIDATE(H)
 for i ← 0 to D(n[H])
 do A[i] ← NIL
 for mỗi nút w trong danh sách gốc của H
      do x ← w
         d ← degree[x]
         while A[d] ≠ NIL
            do y ← A[d]
               if key[x] > key[y]
                  then trao đổi x ↔ y
               FIB-HEAP-LINK(H, y, x)
               A[d] ← NIL
               d ← d+1
         A[d] ← x
 min[H] ← NIL
 for i ← 0 to D(n[H])
      do if A[i] ≠ NIL
            then cộng A[i] vào danh sách gốc của H
               if min[H] = NIL hoặc key[A[i]] < key[min[H]]
                  then min[H] ← A[i]
```

FIB-HEAP-LINK(*H*, *y*, *x*)

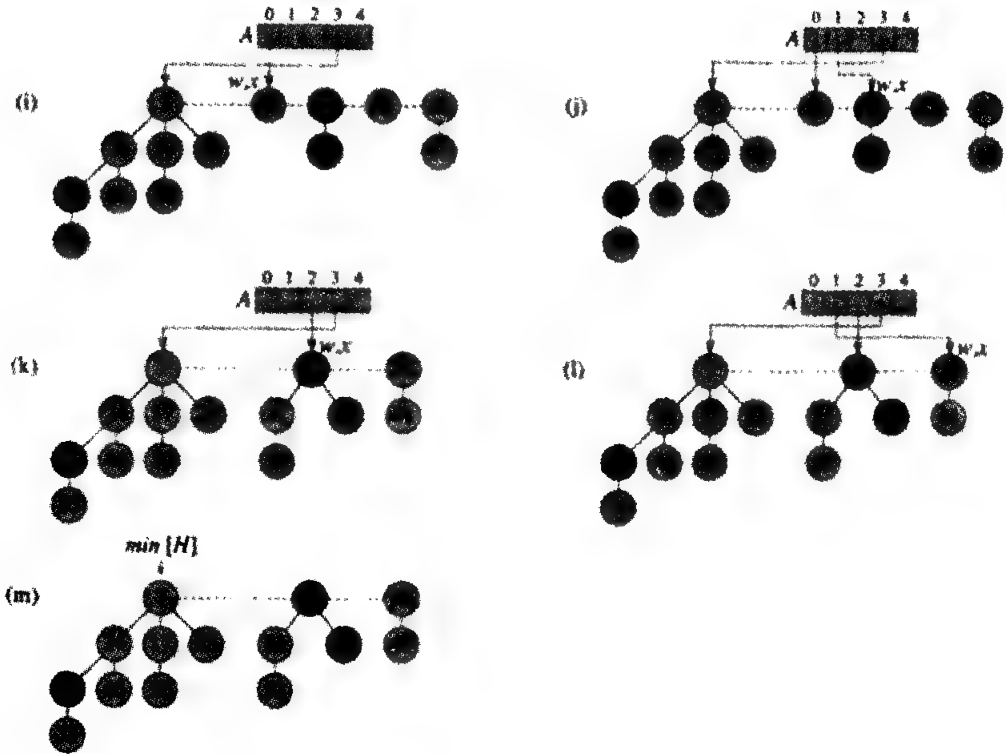
gỡ bỏ *y* ra khỏi danh sách gốc của *H*

chuyển *y* thành một con của *x*, gia số *degree*[*x*]

mark[*y*] ← FALSE



Hình 21.3 Hành động của FIB-HEAP-EXTRACT-MIN. **(a)** Một đống Fibonacci *H*. **(b)** Tình huống sau khi nút cực tiểu *z* được gỡ bỏ ra khỏi danh sách gốc và các con của nó được bổ sung vào danh sách gốc. **(c)**-**(*e*)** Mảng *A* và các cây sau mỗi trong số ba lần lặp lại đầu tiên của vòng lặp **for** của các dòng 3-13 của thủ tục CONSOLIDATE. Danh sách gốc được xử lý bằng cách bắt đầu tại nút cực tiểu và theo các biến trỏ *right*. Mỗi phần nêu các giá trị của *w* và *x* tại cuối của một lần lặp lại. **(*f*)**-**(h)** Lần lặp lại kế tiếp của vòng lặp **for**, có các giá trị của *w* và *x* nêu tại cuối mỗi lần lặp lại của vòng lặp **while** của các dòng 6-12. Phần **(*f*)** nêu tình huống sau lần đầu tiên đi qua vòng lặp **while**. Nút có khóa 23 đã được nối kết với nút có khóa 7, giờ đây được *x* trỏ đến. Trong phần **(*g*)**, nút có khóa 17 đã được nối kết với nút có khóa 7, vẫn được *x* trỏ đến. Trong phần **(h)**, nút có khóa 24 đã được nối kết với nút có khóa 7. Do không có nút nào được *A*[3] trỏ đến trước đó, nên tại cuối lần lặp lại của vòng lặp **for**, *A*[3] được ấn định trỏ đến gốc của cây kết quả. **(*i*)**-**(l)** Tình huống sau mỗi trong số bốn lần lặp lại kế tiếp của vòng lặp **while**. **(*m*)** Đống Fibonacci *H* sau khi kiến tạo lại danh sách gốc từ mảng *A* và xác định biến trỏ mới *min*[*H*].



Về chi tiết, thủ tục CONSOLIDATE làm việc như sau. Trong các dòng 1-2, ta khởi tạo *A* bằng cách chuyển mỗi khoản nhập thành NIL. Khi ta hoàn tất tiến trình xử lý từng gốc *w*, nó kết thúc bằng một cây có gốc tại một nút *x*, mà có thể hoặc không đồng nhất với *w*. Sau đó, khoản nhập mảng *A*[*degree*[*x*]] sẽ được ấn định để trỏ đến *x*. Trong vòng lặp **for** của các dòng 3-13, ta xét từng gốc *w* trong danh sách gốc. Sự bất biến được duy trì trong mỗi lần lặp lại của vòng lặp **for** đó là nút *x* là gốc của cây chứa nút *w*. Vòng lặp **while** của các dòng 6-12 duy trì sự bất biến *d* = *degree*[*x*] (ngoại trừ trong dòng 11, như sẽ thấy dưới đây). Trong mỗi lần lặp lại của vòng lặp **while**, *A*[*d*] trỏ đến một gốc *y*. Do *d* = *degree*[*x*] = *degree*[*y*], nên ta muốn nối kết *x* và *y*. Với *x* và *y* bất kỳ cái nào có khóa nhỏ hơn đều trở thành cha của cái kia như là một kết quả của phép toán nối kết, và do đó các dòng 8-9 trao đổi các biến trỏ đến *x* và *y* nếu cần. Kế đó, ta nối kết *y* với *x* bằng lệnh gọi FIB-HEAP-LINK(*H*, *y*, *x*) trong dòng 10. Lệnh gọi này gia số *degree*[*x*] nhưng để *degree*[*y*] dưới dạng *d*. Bởi nút *y* không còn là một gốc, biến trỏ đến nó trong mảng *A* được gỡ bỏ trong dòng 11. Do giá trị của *degree*[*x*] được gia số bằng lệnh gọi FIB-HEAP-LINK, dòng 12 phục hồi sự bất biến đó là *d* = *degree*[*x*]. Ta lặp lại vòng lặp **while** cho đến khi *A*[*d*] = NIL, trong trường hợp đó không có gốc nào khác có cùng độ như *x*. Ta ấn định

$A[d]$ là x trong dòng 13 và thực hiện lần lặp lại kế tiếp của vòng lặp **for**. Các Hình 21.3(c)-(*e*) nêu mảng A và các cây kết quả sau ba lần lặp lại đầu tiên của vòng lặp **for** của các dòng 3-13. Trong lần lặp lại kế tiếp của vòng lặp **for**, ta có ba nối kết xảy ra; kết quả của chúng được nêu trong các Hình 21.3(*f*)-(h). Các Hình 21.3(*i*)-(1) 1 nêu kết quả của bốn lần lặp lại kế tiếp của vòng lặp **for**.

Tất cả những gì còn lại đó là dọn dẹp. Một khi vòng lặp **for** của các dòng 3-13 hoàn tất, dòng 14 xả trống danh sách gốc, và các dòng 15-19 kiến tạo lại nó. Đống Fibonacci kết quả được nêu trong Hình 21.3(*m*). Sau khi thống nhất danh sách gốc, FIB-HEAP-EXTRACT-MIN hoàn tất bằng cách giảm lượng $n[H]$ trong dòng 11 và trả về một biến trỏ đến nút bị xóa z trong dòng 12.

Nhận thấy nếu tất cả các cây trong đống Fibonacci là các cây nhị thức không sắp xếp trước khi FIB-HEAP-EXTRACT-MIN thi hành, thì tất cả chúng đều là các cây nhị thức không sắp xếp sau đó. Có hai cách làm thay đổi các cây. Thứ nhất, trong các dòng 3-5 của FIB-HEAP-EXTRACT-MIN, mỗi con x của gốc z trở thành một gốc. Với Bài tập 21.2-2, mỗi cây mới chính là một cây nhị thức không sắp xếp. Thứ hai, các cây chỉ được FIB-HEAP-LINK nối kết nếu chúng có cùng độ. Bởi tất cả các cây đều là những cây nhị thức không sắp xếp trước khi nối kết xảy ra, nên hai cây mà từng gốc của chúng có k con sẽ phải có cấu trúc của U_k. Do đó, cây kết quả có cấu trúc của U_{k+1}.

Giờ đây ta đã sẵn sàng chứng tỏ mức hao phí khấu trừ của tiến trình trích nút cực tiểu của một đống Fibonacci n-nút là $O(D(n))$. Cho H thể hiện đống Fibonacci ngay trước phép toán FIB-HEAP-EXTRACT-MIN.

Mức hao phí thực tế của việc trích nút cực tiểu có thể được giải trình như sau. Một khoản đóng góp $O(D(n))$ xuất xứ từ nơi có tối đa $D(n)$ con của nút cực tiểu được xử lý trong FIB-HEAP-EXTRACT-MIN và từ công việc trong các dòng 1-2 và 14-19 của CONSOLIDATE. Việc còn lại đó là phân tích phần đóng góp từ vòng lặp **for** của các dòng 3-13. Kích cỡ của danh sách gốc vào lúc gọi CONSOLIDATE tối đa là $D(n) + t(H) - 1$, bởi nó bao gồm $t(H)$ nút danh sách gốc ban đầu, trừ cho nút gốc đã trích, cộng với các con của nút đã trích, mà số tối đa là $D(n)$. Mọi lần qua vòng lặp **while** của các dòng 6-12, một trong các gốc được nối kết với một gốc khác, và như vậy tổng lượng công việc được thực hiện trong vòng lặp **for** tối đa tỷ lệ với $D(n) + t(H)$. Như vậy, tổng công việc thực tế là $O(D(n) + t(H))$.

Thế trước khi trích nút cực tiểu là $t(H) + 2\,m(H)$, và thế sau đó tối đa là $(D(n) + 1) + 2\,m(H)$, bởi có tối đa $D(n) + 1$ gốc giữ nguyên và không

có nút nào được đánh dấu trong phép toán. Như vậy, mức hao phí khấu trừ tối đa là

$$O(D(n) + t(H)) + ((D(n) + 1) + 2\,m(H)) - (t(H) + 2\,m(H))$$
$$= O(D(n)) + O(t(H)) - t(H)$$
$$= O(D(n))\,,$$

bởi ta có thể nâng các đơn vị của thế để chi phối hằng ẩn trong $O(t(H))$. Theo trực giác, mức hao phí để thực hiện từng nối kết được thanh toán bởi phép giảm thế, do nối kết đã rút gọn số lượng gốc xuống một.

Bài tập

21.2-1

Nêu đống Fibonacci là kết quả của việc gọi FIB-HEAP-EXTRACT-MIN trên đống Fibonacci nêu trong Hình 21.3(*m*).

21.2-2

Chứng minh Bổ đề 20.1 áp dụng cho các cây nhị thức không sắp xếp, nhưng với tính chất 4' thay vì tính chất 4.

21.2-3

Chứng tỏ nếu chỉ hỗ trợ các phép toán đống khả trộn, độ cực đại $D(n)$ trong một đống Fibonacci n-nút tối đa là $\lfloor \lg n \rfloor$.

21.2-4

Giáo sư McGee đã nghĩ ra một cấu trúc dữ liệu mới dựa trên các đống Fibonacci. Một đống McGee có cùng cấu trúc như một đống Fibonacci và hỗ trợ các phép toán đống khả trộn. Các kiểu thực thi của các phép toán giống như trường hợp các đống Fibonacci, ngoại trừ phép chèn và hợp thực hiện tiến trình thống nhất làm bước cuối của chúng. Đâu là thời gian thực hiện ca xấu nhất của các phép toán trên các đống McGee? Cấu trúc dữ liệu của giáo sư mới lạ chỗ nào?

21.2-5

Chứng tỏ khi các phép toán duy nhất trên các khóa đang so sánh hai khóa (như trường hợp với tất cả các kiểu thực thi trong chương này), không phải tất cả các phép toán đống khả trộn đều có thể chạy trong $O(1)$ thời gian khấu trừ.

21.3 Giảm một khóa và xóa một nút

Trong đoạn này, ta nêu cách giảm khóa của một nút trong một đống Fibonacci trong $O(1)$ thời gian khấu trừ và cách xóa một nút bất kỳ ra khỏi một đống Fibonacci n-nút trong $O(D(n))$ thời gian khấu trừ. Các phép toán này không bảo toàn tính chất cho rằng tất cả các cây trong đống Fibonacci là các cây nhị thức không sắp xếp. Tuy nhiên, chúng đủ sát để ta có thể định cận độ cực đại $D(n)$ theo $O(\lg n)$. Việc chứng minh cận này sẽ hàm ý rằng FIB-HEAP-EXTRACT-MIN và FIB-HEAP-DELETE chạy trong $O(\lg n)$ thời gian khấu trừ.

Giảm một khóa

Cũng như trước đây, trong mã giả dưới đây của phép toán FIB-HEAP-DECREASE-KEY, ta mặc nhận việc gỡ bỏ một nút ra khỏi một danh sách nối kết sẽ không làm thay đổi bất kỳ trong số các trường có cấu trúc trong nút được gỡ bỏ.

```
FIB-HEAP-DECREASE-KEY(H, x, k)
if k > key[x]
      then error "khóa mới lớn hơn khóa hiện hành"
key[x] ← k
y ← p[x]
if y ≠ NIL và key[x] < key[y]
      then CUT (H, x, y)
           CASCADING-CUT(H, y)
if key[x] < key[min[H]]
      then min[H] ← x

CUT(H, x, y)
gỡ bỏ x ra khỏi danh sách con của y, giảm lượng degree[y]
cộng x vào danh sách gốc của H
p[x] ← NIL
mark[x] ← FALSE

CASCADING-CUT(H, y)
z ← p[y]
if z ≠ NIL
    then if mark[y] = FALSE
```

```
4            then mark[y] ← TRUE
5            else CUT (H, y, z)
6                 CASCADING-CUT(H, z)
```

Thủ tục FIB-HEAP-DECREASE-KEY làm việc như sau. Các dòng 1-3 bảo đảm khóa mới không lớn hơn khóa hiện hành của x và sau đó gán khóa mới cho x. Nếu x là một gốc hoặc giả $key[x] \geq key[y]$, ở đó y là cha của x, thì không cần thay đổi về cấu trúc, bởi thứ tự đống không bị vi phạm. Các dòng 4-5 kiểm tra điều kiện này.

Nếu thứ tự đống đã bị vi phạm, nhiều thay đổi có thể xảy ra. Ta bắt đầu bằng cách ***cắt*** x trong dòng 6. Thủ tục CUT "cắt" nối kết giữa x và cha y của nó, biến x thành một gốc.

Ta dùng các trường *mark* để được các cận thời gian mong muốn. Chúng giúp tạo hiệu ứng dưới đây. Giả sử x là một nút đã trải qua lịch sử dưới đây:

1. tại một thời gian nào đó, x là một gốc,

2. thì x đã được nối kết với một nút khác,

3. thì hai con của x đã được gỡ bỏ bởi các tác vụ cắt.

Ngay khi con thứ hai bị mất, x được cắt ra khỏi cha của nó, biến nó thành một gốc mới. Trường *mark*[x] là TRUE nếu các bước 1 và 2 đã xảy ra và một con của x đã được cắt. Do đó thủ tục CUT xóa *mark*[x] trong dòng 4, bởi nó thực hiện bước 1. (Giờ đây ta có thể thấy tại sao dòng 3 của FIB-HEAP-LINK lại xóa *mark*[y]: nút y đang được nối kết với một nút khác, và do đó bước 2 đang được thực hiện. Thời gian kế tiếp mà một con của y được cắt, *mark*[y] sẽ được ấn định là TRUE.)

Ta chưa hoàn tất, bởi x có thể là con thứ hai được cắt ra khỏi cha y của nó từ lúc mà y đã được nối kết với một nút khác. Do đó, dòng 7 của FIB-HEAP-DECREASE-KEY thực hiện một phép toán ***cắt liên hoàn*** [cascading-cut] trên y. Nếu y là một gốc, đợt kiểm tra trong dòng 2 của CASCADING-CUT sẽ khiến thủ tục đơn giản trở về. Nếu y không được đánh dấu, thủ tục đánh dấu nó trong dòng 4, bởi con đầu tiên của nó vừa được cắt, và trở về. Tuy nhiên, nếu y được đánh dấu, nó vừa bị mất con thứ hai của nó; y được cắt trong dòng 5, và CASCADING-CUT tự gọi nó một cách đệ quy trong dòng 6 trên cha z của y. Thủ tục CASCADING-CUT đệ quy theo hướng lên cây cho đến khi tìm thấy một gốc hoặc một nút không đánh dấu.

Sau khi xảy ra tất cả các tác vụ cắt liên hoàn, các dòng 8-9 của FIB-HEAP-DECREASE-KEY hoàn tất bằng cách cập nhật *min*[*H*] nếu cần.

Hình 21.4 nêu việc thi hành hai lệnh gọi của FIB-HEAP-DECREA.SEKEY, bắt đầu với đống Fibonacci nêu trong Hình 21.4(a). Lệnh gọi đầu tiên, nêu trong Hình 21.4(b), không liên quan đến các tác vụ cắt liên hoàn. Lệnh gọi thứ hai, nêu trong các Hình 21.4(c)-(*e*), triệu gọi hai tác vụ cắt liên hoàn.

Giờ đây, ta chứng tỏ mức hao phí khấu trừ của FIB-HEAP-DECREASEKEY chỉ là $O(1)$. Ta bắt đầu bằng cách xác định mức hao phí thực tế của nó. Thủ tục FIB-HEAP-DECREASE-KEY mất $O(1)$ thời gian, cộng với thời gian thực hiện các tác vụ cắt liên hoàn. Giả sử CASCADING-CUT được gọi theo đệ quy c lần từ một lệnh triệu mời FIB-HEAP-DECREASE-KEY đã cho. Mỗi lệnh gọi CASCADING-CUT mất $O(1)$ thời gian không kể các lệnh gọi đệ quy. Như vậy, mức hao phí thực tế của FIB-HEAP-DECREASE-KEY, kể cả tất cả các lệnh gọi đệ quy, là $O(c)$.

Kế đó, ta tính sự thay đổi trong thế. Cho H thể hiện đống Fibonacci ngay trước phép toán FIB-HEAP-DECREASE-KEY. Mỗi lệnh gọi đệ quy của CASCADING-CUT, ngoại trừ lệnh cuối, sẽ cắt một nút có đánh dấu và xóa bit dấu. Sau đó, ta có $t(H) + c$ cây ($t(H)$ cây ban đầu, c- 1 cây do các tác vụ cắt liên hoàn tạo ra, và cây có gốc tại x) và tối đa $m(H) - c + 2$ nút có đánh dấu (c - 1 đã được các tác vụ cắt liên hoàn thôi đánh dấu và lệnh gọi CASCADING-CUT cuối có thể đã đánh dấu một nút). Do đó, thay đổi trong thế tối đa là

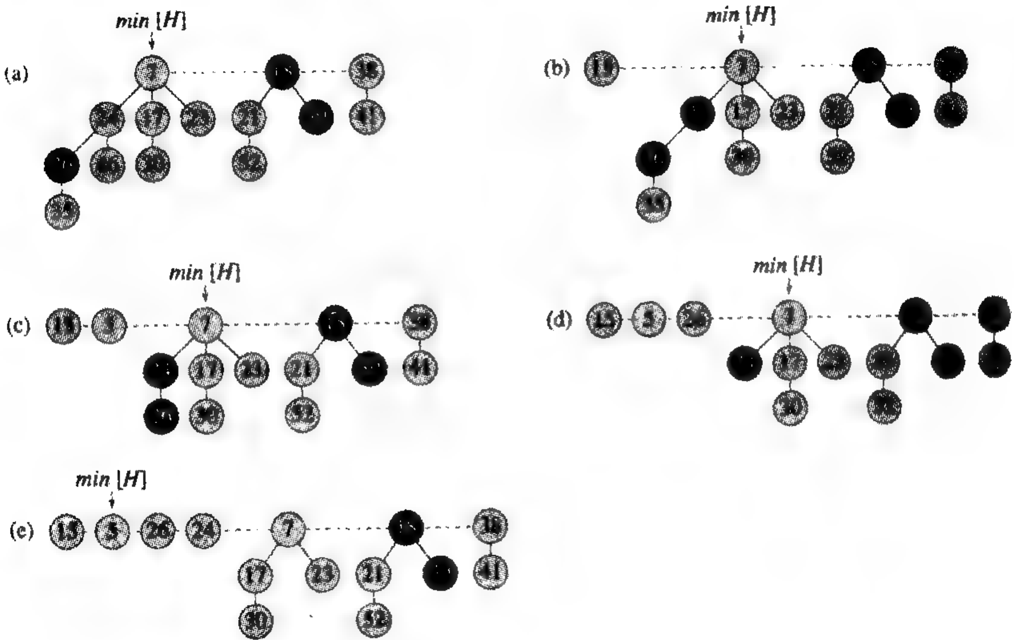
$$((t(H) + c) + 2(m(H) - c + 2)) - (t(H) + 2\,m(H)) = 4 - c\ .$$

Như vậy, mức hao phí khấu trừ của FIB-HEAP-DECREASE-KEY tối đa là

$$O(c) + 4 - c = O(1),$$

bởi ta có thể nâng các đơn vị của thế để chi phối hằng ẩn trong $O(c)$.

Giờ đây, bạn có thể thấy tại sao hàm thế lại được định nghĩa để gộp một số hạng gấp hai lần số lượng các nút được đánh dấu. Khi một nút đánh dấu y được cắt bởi một tác vụ cắt liên hoàn, bit dấu của nó được xóa, do đó thế được rút gọn đi 2. Một đơn vị của thế thanh toán cho tác vụ cắt và xóa bit dấu, và đơn vị kia bù cho đơn vị gia tăng trong thế do nút y trở thành một gốc.



Hình 21.4 Hai lệnh gọi FIB-HEAP-DECREASE-KEY. **(a)** Đống Fibonacci ban đầu. **(b)** Nút có khóa 46 đã giảm khóa của nó thành 15. Nút trở thành một gốc, và cha của nó (có khóa 24), mà trước đó đã được xóa dấu, nay trở thành được đánh dấu. **(c)-(e)** Nút có khóa 35 giảm khóa của nó thành 5. Trong phần (c), nút, mà giờ đây có khóa 5, trở thành một gốc. Cha của nó, có khóa 26, được đánh dấu, do đó một tác vụ cắt liên hoàn xảy ra. Nút có khóa 26 được cắt ra khỏi cha của nó và tạo một gốc xóa dấu trong (d). Một tác vụ cắt liên hoàn khác xảy ra, bởi nút có khóa 24 cũng được đánh dấu. Nút này được cắt ra khỏi cha của nó và tạo một gốc xóa dấu trong phần (*e*). Các tác vụ cắt liên hoàn dừng tại điểm này, bởi nút có khóa 7 là một gốc. (Cho dù nút này không phải là một gốc, các tác vụ cắt liên hoàn cũng sẽ dừng, bởi nó được xóa dấu.) Kết quả của phép toán FIB-HEAP-DECREASE-KEY được nêu trong phần (*e*), với *min*[*H*] trỏ đến nút cực tiểu mới.

Xóa một nút

Ta có thể dễ dàng xóa một nút ra khỏi một đống Fibonacci n-nút trong $O(D(n))$ thời gian khấu trừ, như trong trường hợp của mã giả dưới đây. Ta mặc nhận rằng không có giá trị khóa nào của $-\infty$ hiện nằm trong đống Fibonacci.

```
FIB-HEAP-DELETE(H, x)
FIB-HEAP-DECREASE-KEY(H, x, -∞)
FIB-HEAP-EXTRACT-MIN(H)
```

FIB-HEAP-DELETE tương tự như BINOMIAL-HEAP-DELETE. Nó

biến x trở thành nút cực tiểu trong đống Fibonacci bằng cách gán cho nó một khóa nhỏ duy nhất của $-\infty$. Sau đó, nút x được thủ tục FIB-HEAP-EXTRACT-MIN gỡ bỏ ra khỏi đống Fibonacci. Thời gian khấu trừ của FIB-HEAPDELETE là tổng thời gian khấu trừ $O(1)$ của FIB-HEAP-DECREASEKEY và thời gian khấu trừ $O(D(n))$ của FIB-HEAP-EXTRACT-MIN.

Bài tập

21.3-1

Giả sử một gốc x trong một đống Fibonacci được đánh dấu. Giải thích cách x trở thành một gốc được đánh dấu. Chứng tỏ việc phân tích x được đánh dấu là không thành vấn đề, cho dù nó không phải là một gốc đầu tiên được nối kết với một nút khác và rồi bị mất một con.

21.3-2

Xác minh $O(1)$ thời gian khấu trừ của FIB-HEAP-DECREASE-KEY dùng phương pháp kết tập của Đoạn 18.1.

21.4 Định cận độ cực đại

Để chứng minh thời gian khấu trừ của FIB-HEAP-EXTRACT-MIN và FIB-HEAP-DELETE là $O(\lg n)$, ta phải chứng tỏ cận trên $D(n)$ trên độ của bất kỳ nút nào của một đống Fibonacci n-nút là $O(\lg n)$. Qua Bài tập 21.2-3, khi tất cả các cây trong đống Fibonacci là các cây nhị thức không sắp xếp, $D(n) = \lfloor \lg n \rfloor$. Tuy nhiên, các tác vụ cắt xảy ra trong FIB-HEAP-DECREASEKEY có thể khiến các cây trong đống Fibonacci vi phạm các tính chất của cây nhị thức không sắp xếp. Trong đoạn này, ta sẽ chứng tỏ do ta cắt một nút ra khỏi cha của nó ngay khi nó mất hai con, nên $D(n)$ là $O(\lg n)$. Nói cụ thể, ta sẽ chứng tỏ $D(n) \leq \lfloor \log_\phi n \rfloor$, ở đó $D)=(1 + \sqrt{5})/2$.

Điểm cốt lõi của phân tích là như sau. Với mỗi nút x trong một đống Fibonacci, ta định nghĩa size(x) là số lượng các nút, kể cả chính x, trong cây con có gốc tại x. (Lưu ý, x không buộc phải nằm trong danh sách gốc—nó có thể là một nút bất kỳ.) Ta sẽ chứng tỏ size(x) là hàm mũ trong *degree*[x]. Đừng quên *degree*[x] luôn được duy trì dưới dạng một số đếm chính xác độ của x.

***Bổ đề* 21.1**

Cho x là một nut bất kỳ trong một đống Fibonacci, và giả sử rằng

degree[x] = k. Cho $y_1, y_2,..., y_k$ thể hiện các con của x theo thứ tự mà chúng được nối kết với x, từ sớm nhất đến muộn nhất. Như vậy, *degree*[y_1] $\geq$ 0 và *degree*[y_i] $\geq i - 2$ với $i = 2, 3,..., k$.

Chứng minh Hiển nhiên, *degree*[y_1] $\geq$ *0*.

Với $i \geq 2$, ta lưu ý khi y_i được nối kết với x, tất cả $y_1, y_2,..., y_{i-1}$ là các con của x, do đó ta phải có *degree*[x] $\geq i - 1$. Nút y_i chỉ được nối kết với x nếu *degree*[x] = *degree*[y_i], do đó ta cũng phải có *degree*[y_i] $\geq i - 1$ vào lúc đó. Từ đó, nút y_i đã mất tối đa một con, bởi nó đã được cắt ra khỏi x nếu nó mất hai con. Ta kết luận *degree*[y_i] $\geq i - 2$.

Cuối cùng ta đến phần phân tích giải thích tên "các đống Fibonacci." Hãy nhớ lại trong Đoạn 2.2 rằng với k = 0, 1, 2,..., số Fibonacci thứ k được định nghĩa bởi phép truy toán

$$F_k = \begin{cases} 0 & \text{nếu } k = 0, \\ 1 & \text{nếu } k = 1, \\ F_{k-1} + F_{k-2} & \text{nếu } k \geq 2. \end{cases}$$

Bổ đề dưới đây nêu một cách khác để diễn tả F_k.

Bổ đề 21.2

Với tất cả các số nguyên $k \geq 0$,

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i .$$

Chứng minh Phần chứng minh dùng phương pháp quy nạp trên k. Khi $k = 0$,

$$\begin{aligned} 1 + \sum_{i=0}^{0} F_i &= 1 + F_0 \\ &= 1 + 0 \\ &= 1 \\ &= F_2 . \end{aligned}$$

Giờ đây ta mặc nhận giả thuyết quy nạp rằng $F_{k+1} = 1 + \sum_{i=0}^{k-1} F_i$. và ta có

$$\begin{aligned} F_{k+2} &= F_k + F_{k-1} \\ &= F_k + \left(1 + \sum_{i=0}^{k-1} F_i\right) \\ &= 1 + \sum_{i=0}^{k} F_i . \end{aligned}$$

Bổ đề dưới đây và hệ luận của nó hoàn thành sự phân tích. Nó sử dụng bất đẳng thức (được chứng minh trong Bài tập 2.2-8)

$$F_{k+2} \geq \phi^k ,$$

ở đó 0 là tỷ số vàng được định nghĩa trong phương trình (2.14) vì $\phi = (1 + \sqrt{5})/2 = 1.61803....$

Bổ đề 21.3

Cho x là một nút bất kỳ trong một đống Fibonacci, và cho $k = degree[x]$. Như vậy, size(x) $\geq F_{k+2} \geq \phi^k$, ở đó $\phi = (1 + \sqrt{5})/2$.

Chứng minh Cho s_k thể hiện giá trị cực tiểu khả dĩ có size(z) trên tất cả các nút z sao cho $degree[z] = k$. Bình thường, $s_0 = 1$, $s_1 = 2$, và $s_2 = 3$. Số s_k tối đa là size(x). Như trong Bổ đề 21.1, cho $y_1, y_2, ..., y_k$ thể hiện các con của x theo thứ tự ở đó chúng được nối kết với x. Để tính toán một cận dưới trên size(x), ta đếm một cho chính x và một cho con y_1 đầu tiên (mà size(y_1) $\geq$ 1) rồi áp dụng Bổ đề 21.1 cho các con khác. Như vậy ta có

$$\begin{aligned} \text{size}(x) &\geq s_k \\ &\geq 2 + \sum_{i=2}^{k} s_{i-2} . \end{aligned}$$

Giờ đây, bằng phương pháp quy nạp trên k ta chứng tỏ $s_k \geq F_{k+2}$ với tất cả số nguyên không âm k. Với $k = 0$ và $k = 1$, các cơ số là không quan trọng. Với bước quy nạp, ta mặc nhận $k \geq 2$ và $s_i \geq F_{i+2}$ với $i = 0, 1, ..., k - 1$. Ta có

$$\begin{aligned} s_k &\geq 2 + \sum_{i=2}^{k} s_{i-2} \\ &\geq 2 + \sum_{i=2}^{k} F_i . \\ &= 1 + \sum_{i=0}^{k} F_i \\ &= F_{k+2} . \end{aligned}$$

Đẳng thức cuối là do Bổ đề 21.2.

Như vậy, ta đã chứng tỏ size(x) $\geq s_k \geq F_{k+2} \geq \phi^k$.

Hệ luận 21.4

Độ cực đại $D(n)$ của bất kỳ nút nào trong một đống Fibonacci n-nút là $O(\lg n)$.

Chứng minh Cho x là một nút bất kỳ trong một đống Fibonacci n-nút,

và cho $k = degree[x]$. Theo Bổ đề 21.3, ta có $n \geq \text{size}(x) \geq \phi^k$. Lấy lôga cơ số ϕ cho ra $k \leq \log_\phi n$. (Thực vậy, do k là một số nguyên, $k \leq \lfloor \log_\phi n \rfloor$. Như vậy, độ cực đại $D(n)$ của một nút bất kỳ là $O(\lg n)$.

Bài tập

21.4-1

Giáo sư Pinocchio cho rằng chiều cao của một đống Fibonacci n-nút là $O(\lg n)$. Chứng tỏ giáo sư bị lầm khi biểu lộ, với bất kỳ số nguyên dương n, một dãy các phép toán đống Fibonacci tạo một đống Fibonacci chỉ gồm một cây là một xích tuyến tính gồm n nút.

21.4-2

Giả sử ta tổng quát hóa quy tắc cắt liên hoàn để cắt một nút x ra khỏi cha của nó ngay khi nó mất con thứ k của nó, với một hằng số nguyên k. (Quy tắc trong Đoạn 21.3 sử dụng $k = 2$.) Với các giá trị nào của k, $D(n) = O(\lg n)$?

Bài Toán

21-1 Cách thực thi khác của phép xóa

Giáo sư Pisano đã đề xuất biến thể dưới đây của thủ tục FIB-HEAPDELETE, cho rằng nó chạy nhanh hơn khi nút được xóa không phải là nút được trỏ đến bởi $min[H]$.

```
PISANO-DELETE(H, x)
1  if x = min[H]
2     then FIB-HEAP-EXTRACT-MIN(H)
3     else y ← p[x]
4           if y ≠ NIL
5              then CUT (H, x, y)
6                    CASCADING-CUT(H, y)
7           cộng danh sách con của x vào danh sách gốc của H
8           gỡ bỏ x ra khỏi danh sách gốc của H
```

a. Khi nói thủ tục này chạy nhanh hơn, giáo sư đã phần nào dựa trên giả thiết cho rằng dòng 7 có thể được thực hiện trong $O(1)$ thời gian thực tế. Giả thiết này có sai?

b. Nêu một cận trên thích hợp trên thời gian thực tế của PISANO-

DELETE khi $x \neq min[H]$. Cận của bạn phải theo dạng *degree*[*x*] và con số *c* lệnh gọi đến thủ tục CASCADING-CUT.

c. Cho *H'* là đống Fibonacci xuất xứ từ một đợt thi hành của PISANO-DELETE(*H*, *x*). Giả sử nút *x* không phải là một gốc, hãy định cận thế của *H'* theo dạng *degree*[*x*], *c*, *t*(*H*), và *m*(*H*).

d. Kết luận rằng thời gian khấu trừ của PISANO-DELETE theo tiệm cận không tốt hơn cho FIB-HEAP-DELETE, thậm chí khi $x \neq min[H]$.

21-2 Thêm các phép toán đống Fibonacci

Ta muốn tăng cường một đống Fibonacci *H* để hỗ trợ hai phép toán mới mà không làm thay đổi thời gian thực hiện khấu trừ của bất kỳ các phép toán đống Fibonacci nào khác.

a. Nêu một kiểu thực thi hiệu quả của phép toán FIB-HEAP-CHANGE-KEY(*H*, *x*, *k*), thay đổi khóa của nút *x* thành giá trị *k*. Phân tích thời gian thực hiện khấu trừ của cách thực thi của bạn cho các trường hợp ở đó *k* lớn hơn, nhỏ hơn, hoặc bằng với *key*[*x*].

b. Nêu một kiểu thực thi hiệu quả của FIB-HEAP-PRUNE(*H*, *r*), sẽ xóa min(*r*, *n* [*H*]) nút ra khỏi *H*. Những nút nào được xóa điều đó là tùy ý. Phân tích thời gian thực hiện khấu trừ của kiểu thực thi của bạn. (*Mách nước:* Bạn có thể cần sửa đổi cấu trúc dữ liệu và hàm thế.)

Ghi chú Chương

Các đống Fibonacci đã được Fredman và Tarjan [75] giới thiệu. Tài liệu của họ cũng mô tả ứng dụng của các đống Fibonacci đối với các bài toán về lộ trình ngắn nhất một nguồn, các cặp ngắn nhất của tất cả các cặp, so khớp hai nhánh gia trọng, và bài toán cây tỏa nhánh cực tiểu.

Và sau đó, Driscoll, Sarnak, Sleator, và Tarjan [58] đã phát triển "các đống nới lỏng" như một phương án thay thế cho các đống Fibonacci. Có hai biến thể của các đống nới lỏng. Một cho ra các cận thời gian khấu trừ tương tự như các đống Fibonacci. Và một cho phép DECREASE-KEY chạy trong $O(1)$ thời gian trường hợp xấu nhất (chứ không phải khấu trừ) và EXTRACT-MIN và DELETE chạy trong $O(\lg n)$ thời gian trường hợp xấu nhất. Các đống nới lỏng cũng có vài ưu điểm so với các đống Fibonacci trong các thuật toán song song.

22 Các Cấu Trúc Dữ liệu cho Các Tập hợp Rời Nhau

Một số ứng dụng có dính líu đến tiến trình gom nhóm n thành phần riêng biệt thành một tập hợp gồm các tập hợp rời. Như vậy, ta có hai phép toán quan trọng đó là tìm tập hợp chứa một thành phần đã cho và hợp nhất hai tập hợp. Chương này khảo sát các phương pháp để duy trì một cấu trúc dữ liệu hỗ trợ các phép toán này.

Đoạn 22.1 mô tả các phép toán được hỗ trợ bởi một cấu trúc dữ liệu tập hợp rời và trình bày một ứng dụng đơn giản. Trong Đoạn 22.2, ta xét một kiểu thực thi danh sách nối kết đơn giản cho các tập hợp rời. Đoạn 22.3 trình bày một phần biểu diễn hiệu quả hơn dùng các cây có gốc. Với mọi mục tiêu thực tiễn, thời gian thực hiện dùng phần biểu diễn cây là tuyến tính song về lý thuyết nó là siêu tuyến tính [superlinear]. Đoạn 22.4 định nghĩa và mô tả hàm Ackermann và phép nghịch đảo bành trướng rất chậm của nó, xuất hiện trong thời gian thực hiện của các phép toán trên kiểu thực thi dựa trên cây, rồi sử dụng phân tích khấu trừ để chứng minh một cận trên hơi yếu hơn trên thời gian thực hiện.

22.1 Các phép toán tập hợp rời

Một ***cấu trúc dữ liệu tập hợp rời*** [disjoint-data structure] duy trì một tập hợp $S = \{S_1, S_2, ..., S_k\}$ gồm các tập hợp động rời nhau. Mỗi tập hợp được định danh bởi một ***đại diện*** [representative], là vài phần tử của tập hợp. Trong vài ứng dụng, ta không quan tâm phần tử nào được dùng làm đại biểu; ta chỉ quan tâm nếu như yêu cầu đại biểu của một tập hợp động hai lần mà không sửa đổi tập hợp giữa các lần yêu cầu, ta có cùng đáp án cho cả hai lần. Trong các ứng dụng khác, có thể có một quy tắc định sẵn để chọn đại biểu, như chọn phần tử nhỏ nhất trong tập hợp (tất nhiên, mặc nhận các thành phần có thể được sắp xếp).

Như trong các kiểu thực thi tập hợp động khác mà ta đã nghiên cứu, mỗi thành phần của một tập hợp được biểu thị bởi một đối tượng. Cho x thể hiện một đối tượng, ta muốn hỗ trợ các phép toán sau đây.

MAKE-SET(x) tạo một tập hợp mới mà phần tử duy nhất của nó (và

do đó là đại biểu) được trỏ đến bởi *x*. Do các tập hợp rời nhau, nên ta yêu cầu *x* chưa nằm trong một tập hợp.

UNION(*x*, *y*) hợp nhất các tập hợp động chứa *x* và *y*, giả sử S_x và S_y, thành một tập hợp mới là hợp của hai tập hợp đó. Hai tập hợp được mặc nhận rời nhau trước phép toán. Đại biểu của tập hợp kết quả là vài phần tử của $S_x \cup S_y$, mặc dù có nhiều kiểu thực thi của UNION chọn đại biểu của S_x hoặc S_y làm đại biểu mới. Bởi ta yêu cầu các tập hợp trong tập hợp rời nhau, nên ta "hủy" các tập hợp S_x và S_y, gỡ bỏ chúng ra khỏi tập hợp *S*.

FIND-SET(*x*) trả về một biến trỏ đến đại biểu của tập hợp (duy nhất) chứa *x*.

Suốt chương này, ta sẽ phân tích các thời gian thực hiện của các cấu trúc dữ liệu tập hợp rời theo dạng hai tham số: *n*, số lượng các phép toán MAKE-SET, và *m*, tổng các phép toán MAKE-SET, UNION, và FIND-SET. Bởi các tập hợp rời nhau, nên mỗi phép toán UNION sẽ rút gọn số lượng các tập hợp đi một. Do đó, sau $n - 1$ phép toán UNION, ta chỉ còn lại một tập hợp. Như vậy, số lượng các phép toán UNION tối đa là $n - 1$. Cũng lưu ý, do các phép toán MAKE-SET được gộp trong tổng các phép toán *m*, nên ta có $m \geq n$.

Một ứng dụng về các cấu trúc dữ liệu tập hợp rời

Một trong nhiều ứng dụng của các cấu trúc dữ liệu tập hợp rời nảy sinh trong khi xác định các thành phần liên thông của một đồ thị không có hướng (xem Đoạn 5.4). Ví dụ, Hình 22.1(a) nêu một đồ thị gồm bốn thành phần liên thông.

Thủ tục CONNECTED-ELEMENTS dưới đây sử dụng các phép toán tập hợp rời để tính toán các thành phần liên thông của một đồ thị. Một khi CONNECTED-ELEMENTS chạy dưới dạng một bước tiền xử lý, thủ tục SAME-ELEMENT đáp ứng các lệnh truy vấn về việc hai đỉnh có nằm trong cùng thành phần liên thông hay không[1]. (Tập hợp các đỉnh của một đồ thị *G* được ký hiệu bằng *V*[*G*], và tập hợp các cạnh được ký hiệu bằng *E*[*G*].)

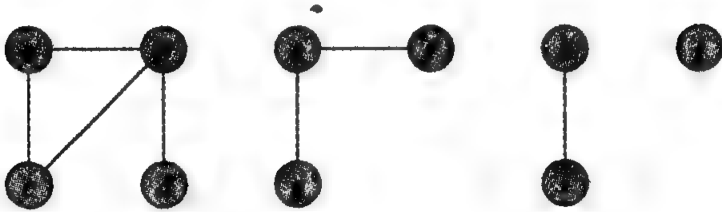
CONNECTED-COMPONENTS (*G*)

1 **for** mỗi đỉnh $v \in V[G]$

---

1 Khi các cạnh của đồ thị là "tĩnh"—không thay đổi qua thời gian—các thành phần liên thông có thể được tính toán nhanh hơn bằng cách dùng kỹ thuật tìm độ sâu đầu tiên (Bài tập 23.3-9). Tuy nhiên, đôi lúc, các cạnh được bổ sung "động" và ta cần duy trì các thành phần liên thông khi bổ sung mỗi cạnh. Trong trường hợp này, kiểu thực thi đã cho ở đây có thể hiệu quả hơn so với việc chạy một đợt tìm kiếm độ sâu đầu tiên cho mỗi cạnh mới.

```
        do MAKE-SET(v)
  for mỗi cạnh (u, v) ∈ E[G]
        do if FIND-SET(u) ≠ FIND-SET(v)
              then UNION(u, v)
```



(a)

Cạnh được xử lý	Tập hợp các tập hợp rời									
các tập hợp ban đầu	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e,g)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a,c)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h,i)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(a,b)	$\{a,b,c,d\}$				$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(e,f)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$
(b,c)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$

(b)

Hình 22.1 **(a)** Một đồ thị có bốn thành phần liên thông: $\{a, b, c, d\}$, $\{e, f, g\}$, $\{h, i\}$, và $\{j\}$. **(b)** Tập hợp các tập hợp rời sau khi xử lý mỗi cạnh.

```
SAME-COMPONENT(u, v)
  if FIND-SET(u) = FIND-SET(v)
      then return TRUE
      else return FALSE
```

Thoạt đầu, thủ tục CONNECTED-COMPONENTS đặt mỗi đỉnh v trong tập hợp riêng của nó. Sau đó, với mỗi cạnh (u, v), nó hợp nhất các tập hợp chứa u và v. Qua Bài tập 22.1-2, sau khi xử lý tất cả các cạnh, hai đỉnh nằm trong cùng thành phần liên thông nếu và chỉ nếu các đối tượng tương ứng nằm trong cùng tập hợp. Như vậy, CONNECTED-

COMPONENTS tính toán các tập hợp sao cho thủ tục SAME-COMPONENT có thể xác định hai đỉnh có nằm trong cùng thành phần liên thông hay không. Hình 22.1(b) minh họa cách tính toán các tập hợp rời của CONNECTED-COMPONENTS.

Bài tập

22.1-1

Giả sử CONNECTED-COMPONENTS chạy trên đồ thị không có hướng $G = (V, E)$, ở đó $V = \{a, b, c, d, e, f, g, h, i, j, k\}$ và các cạnh của E được xử lý theo thứ tự sau: (d, i), (f, k), (g, i), (b, g), (a, h), (i, j), (d, k), (b, j), (d, f), (g, j), (a, e), (i, d). Liệt kê các đỉnh trong mỗi thành phần liên thông sau mỗi lần lặp lại của các dòng 3-5.

22.1-2

Chứng tỏ sau khi CONNECTED-COMPONENTS xử lý tất cả các cạnh, hai đỉnh nằm trong cùng thành phần liên thông nếu và chỉ nếu chúng nằm trong cùng tập hợp.

22.1-3

Trong khi thi hành CONNECTED-COMPONENTS trên một đồ thị không có hướng $G = (V, E)$ có k thành phần liên thông, FIND-SET được gọi bao nhiêu lần? UNION được gọi bao nhiêu lần? Biểu thị các đáp án của bạn theo dạng $|V|$, $|E|$, và k.

22.2 Phần biểu diễn danh sách nối kết của các tập hợp rời

Một cách đơn giản để thực thi một cấu trúc dữ liệu tập hợp rời đó là biểu diễn từng tập hợp bằng một danh sách nối kết. Đối tượng đầu tiên trong mỗi danh sách nối kết được dùng làm đại biểu của tập hợp của nó. Mỗi đối tượng trong danh sách nối kết chứa một phần tử tập hợp, một biến trỏ đến đối tượng chứa phần tử tập hợp kế tiếp, và một biến trỏ quay về đại biểu. Hình 22.2(a) nêu hai tập hợp. Trong mỗi danh sách nối kết, các đối tượng có thể xuất hiện theo một thứ tự bất kỳ (lệ thuộc vào giả thiết của chúng ta cho rằng đối tượng đầu tiên trong mỗi danh sách là đại biểu).

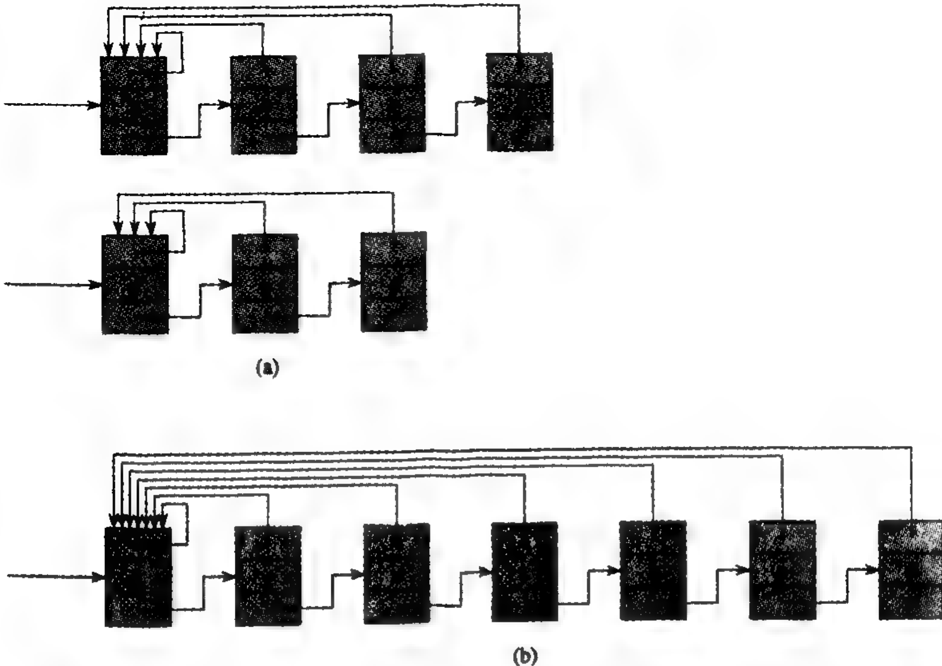
Với phép biểu diễn danh sách nối kết này, cả MAKE-SET lẫn FIND-SET đều dễ dàng, yêu cầu $O(1)$ thời gian. Để thi hành MAKE-SET(x), ta tạo một danh sách nối kết mới mà đối tượng duy nhất của nó là x. Với FIND-SET(x), ta chỉ việc trả biến trỏ từ x trở về đại biểu.

Một thực thi đơn giản của phép hợp

Cách thực thi đơn giản nhất của phép toán UNION dùng phép biểu diễn tập hợp danh sách nối kết kéo dài một thời gian dài hơn đáng kể so với MAKE-SET hoặc FIND-SET. Như Hình 22.2(b) đã nêu, ta thực hiện UNION(x,y) bằng cách chắp danh sách của x vào cuối danh sách của y. Đại biểu của tập hợp mới là thành phần mà ban đầu là đại biểu của tập hợp chứa y. Đáng tiếc, ta phải cập nhật biến trỏ đến đại biểu của từng đối tượng ban đầu nằm trên danh sách của x, nó mất một thời gian tuyến tính theo chiều dài danh sách của x.

Thực vậy, ta dễ dàng gặp một dãy m phép toán yêu cầu $\Theta(m^2)$ thời gian. Ta cho $n = \lceil m/2 \rceil + 1$ và $q = m - n = \lceil m/2 \rceil - 1$ và giả sử có các đối tượng $x_1, x_2, \ldots, x_n$. Sau đó, ta thi hành dãy $m = n + q$ phép toán nêu trong Hình 22.3. Ta bỏ ra $\Theta(n)$ thời gian thực hiện n phép toán MAKE-SET. Do phép toán UNION thứ i cập nhật i đối tượng, nên tổng các đối tượng đã cập nhật bởi tất cả các phép toán UNION sẽ là

$$\sum_{i=1}^{q-1} i = \Theta(q^2) .$$



Hình 22.2 (a) Các kiểu biểu diễn danh sách nối kết của hai tập hợp. Một chứa các đối tượng b, c, e, và h, có c làm đại biểu, và một chứa các đối tượng d, f, và g, có f làm đại biểu. Mỗi đối tượng trên danh sách chứa một phần tử tập hợp, một biến trỏ đến đối tượng kế tiếp trên danh sách, và một biến trỏ quay về đối tượng đầu tiên trên danh sách, là đại biểu. **(b)** Kết quả của *UNION*(e, g). Đại biểu của tập hợp kết quả là f.

Phép toán	Số lượng các đối tượng được cập nhật
MAKE-SET(x_1)	1
MAKE-SET(x_2)	1
.	.
.	.
.	.
MAKE-SET(x_n)	1
UNION(x_1, x_2)	1
UNION(x_2, x_3)	2
UNION(x_3, x_4)	3
.	.
.	.
.	.
UNION(x_{q-1}, x_q)	$q-1$

Hình 22.3 Một dãy m phép toán mất $O(m^2)$ thời gian dùng phép biểu diễn tập hợp danh sách nối kết và kiểu thực thi đơn giản của UNION. Với ví dụ này, $n = \lceil m/2 \rceil + 1$ và $q = m - n$.

Do đó, tổng thời gian bỏ ra là $\Theta(n + q^2)$, chính là $\Theta(m^2)$ bởi $n = \Theta(m)$ và $q = \Theta(m)$. Như vậy, tính trung bình, mỗi phép toán yêu cầu $\Theta(m)$ thời gian. Nghĩa là, thời gian khấu trừ của một phép toán là $\Theta(m)$.

Một heuristic hợp-gia trọng

Kiểu thực thi trên đây của thủ tục UNION yêu cầu một mức trung bình của $\Theta(m)$ thời gian cho mỗi lần gọi bởi có thể ta đang chắp một danh sách dài hơn lên trên một danh sách ngắn hơn; ta phải cập nhật biến trỏ đến đại biểu của mỗi phần tử của danh sách dài hơn. Giả sử thay vì mỗi đại biểu cũng gộp chiều dài của danh sách (dễ duy trì) và ta luôn chắp danh sách nhỏ hơn lên danh sách dài hơn, có các mối ràng buộc được ngắt một cách tùy ý. Với ***heuristic hợp gia trọng*** [weighted-union heuristic] đơn giản này, một phép toán UNION đơn lẻ vẫn có thể mất $\Theta(m)$ thời gian nếu cả hai tập hợp có $\Omega(m)$ phần tử. Tuy nhiên, như định lý dưới đây cho thấy, một dãy m phép toán MAKE-SET, UNION, và FINDSET, n trong số đó là các phép toán MAKE-SET, chiếm $O(m + n \lg n)$ thời gian.

Định lý 22.1

Dùng phép biểu diễn danh sách nối kết của các tập hợp rời và heuristic hợp gia trọng, một dãy m phép toán MAKE-SET, UNION, và FIND-

SET, n trong số đó là các phép toán MAKE-SET, chiếm $O(m + n \lg n)$ thời gian.

Chứng minh Ta bắt đầu bằng cách tính, với mỗi đối tượng trong một tập hợp có kích cỡ n, một cận trên trên số lần đã cập nhật biến trỏ của đối tượng quay về đại biểu. Xét một đối tượng cố định x. Ta biết rằng mỗi lần cập nhật biến trỏ đại biểu của x, x phải bắt đầu trong tập hợp nhỏ hơn. Do đó, lần đầu tiên khi biến trỏ đại biểu của x được cập nhật, tập hợp kết quả phải đã có ít nhất 2 phần tử. Cũng vậy, lần kế tiếp khi biến trỏ đại biểu của x được cập nhật, tập hợp kết quả phải đã có ít nhất 4 phần tử. Tiếp tục như vậy, ta nhận thấy với bất kỳ $k \leq n$, sau khi biến trỏ đại biểu của x đã được cập nhật $\lceil \lg k \rceil$ lần, tập hợp kết quả phải có ít nhất k phần tử. Do tập hợp lớn nhất có tối đa n phần tử, nên biến trỏ đại biểu của mỗi đối tượng đã được cập nhật tối đa $\lceil \lg n \rceil$ lần trên tất cả các phép toán UNION. Như vậy, tổng thời gian được dùng để cập nhật n đối tượng là $O(n \lg n)$.

Thời gian cho nguyên cả dãy m phép toán đi theo dễ dàng. Mỗi phép toán MAKE-SET và FIND-SET mất $O(1)$ thời gian, và có $O(m)$ trong số chúng. Như vậy, tổng thời gian cho nguyên cả dãy là $O(m + n \lg n)$.

Bài tập

22.2-1

Viết mã giả cho MAKE-SET, FIND-SET, và UNION dùng phép biểu diễn danh sách nối kết và heuristic hợp gia trọng. Giả sử mỗi đối tượng x có các thuộc tính *rep*(x] trỏ đến đại biểu của tập hợp chứa x, *last*[x] trỏ đến đối tượng cuối trong danh sách nối kết chứa x, và *size*[x] cho kích cỡ của tập hợp chứa x. Mã giả của bạn có thể mặc nhận rằng *last*[x] và *size*[x] chỉ đúng nếu x là một đại biểu.

22.2-2

Nêu cấu trúc dữ liệu kết quả và các đáp án mà các phép toán FIND-SET trả về trong chương trình sau đây. Dùng phép biểu diễn danh sách nối kết với heuristic hợp gia trọng.

```
for ← 1 to 16
    do MAKE-SET(x_i)
for i ← 1 to 15 by 2
    do UNION(x_i, x_{i+1})
for i ← 1 to 13 by 4
```

```
      do UNION(x_i, x_{i+2})
UNION(x_1, x_5)
UNION(x_11, x_13)
UNION(x_1, x_10)
FIND-SET(x_2)
FIND-SET(x_9)
```

22.2-3

Dựa trên Định lý 22.1, chứng tỏ rằng ta có thể có các cận thời gian khấu trừ là $O(1)$ cho MAKE-SET và FIND-SET và $O(\lg n)$ cho UNION dùng phép biểu diễn danh sách nối kết và heuristic hợp gia trọng.

22.2-4

Nêu một cận tiệm cận chặt trên thời gian thực hiện của dãy các phép toán trong Hình 22.3, mặc nhận phép biểu diễn danh sách nối kết và heuristic hợp gia trọng.

22.3 Các rừng tập hợp rời

Trong một thực thi nhanh hơn của các tập hợp rời, ta biểu diễn các tập hợp bằng các cây có gốc, với mỗi nút chứa một phần tử và mỗi cây biểu thị một tập hợp. Trong một ***rừng tập hợp rời*** [disjoint-set forest], được minh họa trong Hình 22.4(a), mỗi phần tử chỉ trỏ đến cha của nó. Gốc của mỗi cây chứa đại biểu và là cha của riêng nó. Như sẽ thấy, mặc dù các thuật toán đơn giản dùng phép biểu diễn này không nhanh hơn các thuật toán dùng phép biểu diễn danh sách nối kết, song nhờ đưa vào hai heuristic—"hợp theo hạng" và "nén lộ trình"—ta có thể đạt được cấu trúc dữ liệu tập hợp rời nhanh nhất theo tiệm cận đã biết.



Hình 22.4 Một rừng tập hợp rời. **(a)** Hai cây biểu thị hai tập hợp của Hình 22.2. Cây bên trái biểu thị cho tập hợp $\{b, c, e, h\}$, với c làm đại biểu, và cây bên phải biểu thị cho tập hợp $\{d, f, g\}$, với f làm đại biểu. **(b)** Kết quả của UNION(e, g).

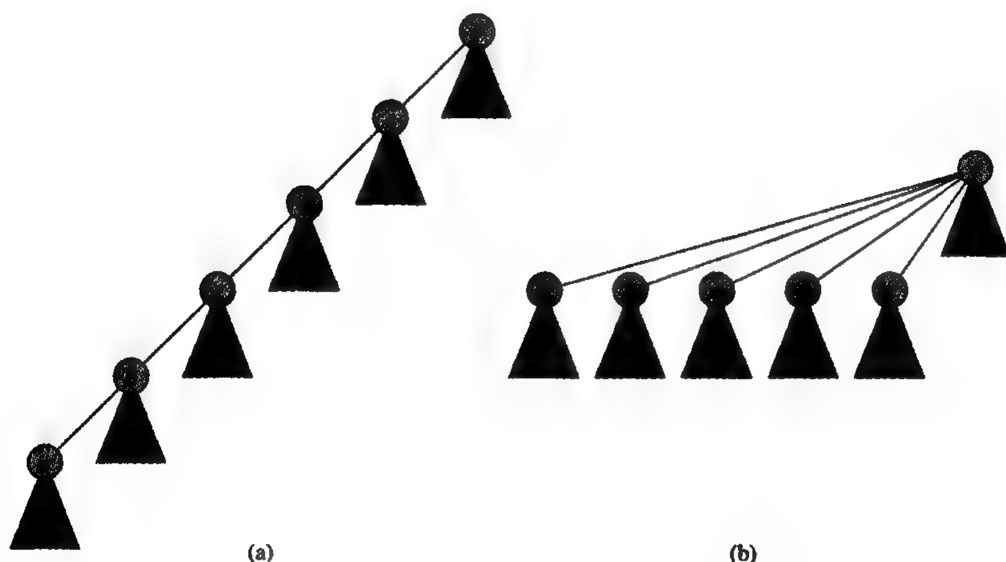
Ta thực hiện ba phép toán tập hợp rời như sau. Một phép toán MAKE-SET đơn giản tạo một cây chỉ có một nút. Ta thực hiện một phép toán FIND-SET bằng cách theo đuổi các biến trỏ cha cho đến khi tìm ra gốc của cây. Các nút đã viếng thăm trên lộ trình này về phía gốc tạo thành ***lộ trình tìm***. Một phép toán UNION, nêu trong Hình 22.4(b), sẽ khiến gốc của một cây trỏ đến gốc của cây kia.

Các heuristic để cải thiện thời gian thực hiện

Cho đến giờ, ta chưa cải thiện kiểu thực thi danh sách nối kết. Một dãy n - 1 phép toán UNION có thể tạo một cây đúng là một xích tuyến tính n nút. Tuy nhiên, nhờ dùng hai heuristic, ta có thể đạt được một thời gian thực hiện hầu như tuyến tính trong tổng các phép toán m.

Heuristic đầu tiên, ***hợp theo hạng*** [union by rank], tương tự như heuristic hợp gia trọng mà ta đã dùng với phép biểu diễn danh sách nối kết. Ý tưởng đó là tạo gốc của cây có ít nút hơn trỏ đến gốc của cây có nhiều nút hơn. Thay vì theo dõi rõ ràng kích cỡ của cây con có gốc tại mỗi nút, ta dùng một cách tiếp cận tạo thuận lợi cho việc phân tích. Với mỗi nút, ta duy trì một ***hạng*** [rank] xấp xỉ với lôga của kích cỡ cây con và cũng là một cận trên trên chiều cao của nút. Trong phương pháp hợp theo hạng, gốc có hạng nhỏ hơn được tạo để trỏ đến gốc có hạng lớn hơn trong một phép toán UNION.

Heuristic thứ hai, ***nén lộ trình*** [path compression], cũng khá đơn giản và rất hiệu quả. Như đã nêu trong Hình 22.5, ta dùng nó trong các phép toán FIND-SET để tạo từng nút trên lộ trình tìm trực tiếp trỏ đến gốc. Phương pháp nén lộ trình không thay đổi các hạng.



Hình 22.5 Nén lộ trình trong phép toán FIND-SET. Các mũi tên và các vòng tự lặp [self-loops] tại các gốc được bỏ qua. **(a)** Một cây biểu thị một tập hợp trước khi thi hành FIND-SET(a). Các tam giác biểu thị cho các cây con có các gốc là các nút được nêu. Mỗi nút có một biến trỏ đến cha của nó. **(b)** Cùng tập hợp sau khi thi hành FIND-SET(a). Mỗi nút trên lộ trình tìm giờ đây trực tiếp trỏ đến gốc.

Mã giả cho các rừng tập hợp rời

Để thực thi một rừng tập hợp rời với heuristic hợp theo hạng, ta phải theo dõi các hạng. Với mỗi nút x, ta duy trì giá trị số nguyên $rank[x]$, là một cận trên trên chiều cao của x (số lượng các cạnh trong lộ trình dài nhất giữa x và một lá hậu duệ). Khi một tập hợp đơn lẻ được MAKE-SET tạo, hạng ban đầu của nút đơn trong cây tương ứng là 0. Mỗi phép toán FIND-SET giữ nguyên tất cả các hạng không đổi. Khi áp dụng UNION cho hai cây, ta biến gốc có hạng cao hơn thành cha của gốc có hạng thấp hơn. Trong trường hợp của một mối ràng buộc [tie], ta tùy ý chọn một trong số các gốc làm cha và gia số hạng của nó.

Ta hãy đưa phương pháp này vào mã giả. Ta chỉ định cha của nút x theo $p[x]$. Thủ tục LINK, một chương trình con được UNION gọi, chấp nhận các biến trỏ đến hai gốc làm đầu vào.

MAKE-SET(x)

1 $p[x] \leftarrow x$

2 $rank[x] \leftarrow 0$

UNION(x,y)

```
1 LINK(FIND-SET(x), FIND-SET(y))
```

```
LINK(x,y)
1 if rank[x] > rank[y]
2     then p[y] ← x
3     else p[x] ← y
4          if rank[x] = rank[y]
5             then rank[y] ← rank[y] + 1
```

Thủ tục FIND-SET với tính năng nén lộ trình khá đơn giản.

```
FIND-SET(x)
1 if x ≠ p[x]
2     then p[x] ← FIND-SET(p[x])
3 return p[x]
```

Thủ tục FIND-SET là một ***phương pháp hai lượt*** [two-pass method]: nó thực hiện một lượt đi lên lộ trình tìm để tìm gốc, và nó thực hiện một lượt thứ hai trở xuống lộ trình tìm để cập nhật mỗi nút để nó trực tiếp trỏ đến gốc. Mỗi lệnh gọi FIND-SET(x) trả về $p[x]$ trong dòng 3. Nếu x là gốc, thì dòng 2 không được thi hành và $p[x] = x$ được trả về. Đây là trường hợp ở đó đệ quy xuống mức thấp nhất. Bằng không, dòng 2 được thi hành, và lệnh gọi đệ quy có tham số $p[x]$ sẽ trả về một biến trỏ đến gốc. Dòng 2 cập nhật nút x để trực tiếp trỏ đến gốc, và biến trỏ này được trả về trong dòng 3.

Hiệu ứng của các heuristic trên thời gian thực hiện

Một cách biệt lập, kỹ thuật hợp theo hạng hoặc nén lộ trình đều cải thiện thời gian thực hiện của các phép toán trên các rừng tập hợp rời, và thậm chí mức cải thiện còn lớn hơn khi hai heuristic được dùng chung với nhau. Một mình, phương pháp hợp theo hạng cho ra cùng thời gian thực hiện như trong trường hợp heuristic hợp gia trọng với phép biểu diễn danh sách: kiểu thực thi kết quả chạy trong thời gian $O(m \lg n)$ (xem Bài tập 22.4-3). Đây là cận chặt (xem Bài tập 22.3-3). Mặc dù ta không chứng minh nó ở đây, song nếu có n phép toán MAKE-SET (và do đó có tối đa n - 1 phép toán UNION) và f phép toán FIND-SET, một mình heuristic nén lộ trình cho ra một thời gian thực hiện trường hợp xấu nhất là $\Theta(f \log_{(1+f/n)} n)$ nếu $f \geq n$ và $\Theta(n + f \lg n)$ nếu $f < n$.

Khi dùng cả heuristic hợp theo hạng lẫn nén lộ trình, thời gian thực hiện trường hợp xấu nhất là $O(m\, \alpha(m, n))$, ở đó $\alpha(m, n)$ là nghịch đảo tăng trưởng *rất* chậm của hàm Ackermann, mà ta định nghĩa trong Đoạn

22.4. Trong bất kỳ ứng dụng nào có thể hình dung được của một cấu trúc dữ liệu tập hợp rời, $\alpha(m, n) \leq 4$; như vậy, ta có thể xem thời gian thực hiện là tuyến tính trong m trong tất cả các tình huống thực tiễn. Trong Đoạn 22.4, ta chứng minh cận hơi yếu hơn của $O(m \lg^* n)$.

Bài tập

22.3-1

Làm Bài tập 22.2-2 dùng một rừng tập hợp rời với heuristic hợp theo hạng và nén lộ trình.

22.3-2

Viết một phiên bản không đệ quy của FIND-SET với tính năng nén lộ trình.

22.3-3

Nêu một dãy m phép toán MAKE-SET, UNION, và FIND-SET, n của nó là các phép toán MAKE-SET, mất $\Omega(m \lg n)$ thời gian khi ta chỉ dùng kỹ thuật hợp theo hạng.

22.3-4*

Chứng tỏ một dãy bất kỳ gồm m phép toán MAKE-SET, FIND-SET, và UNION, ở đó tất cả các phép toán UNION xuất hiện trước mọi phép toán FIND-SET, chỉ mất $O(m)$ thời gian nếu dùng cả phương pháp nén lộ trình lẫn hợp theo hạng. Điều gì xảy ra trong cùng tình huống nếu chỉ dùng heuristic nén lộ trình?

---

* 22.4 Phân tích hợp theo hạng với nén lộ trình

Như đã nêu trong Đoạn 22.3, thời gian thực hiện của heuristic hợp theo hạng và nén lộ trình kết hợp là $O(m\,\alpha(m, n))$ với m phép toán tập hợp rời trên n thành phần. Trong đoạn này, ta sẽ xét hàm α để xem nó tăng trưởng chậm đến mức nào. Sau đó, thay vì trình bày cách chứng minh rất phức tạp của $O(m\,\alpha(m, n))$ thời gian thực hiện, ta sẽ cung cấp một kiểu chứng minh đơn giản hơn của một cận trên hơi yếu hơn trên thời gian thực hiện: $O(m \lg^* n)$.

Hàm của Ackermann và nghịch đảo của nó

Để hiểu rõ hàm Ackermann và nghịch đảo α của nó, ta cần có một hệ ký hiệu cho phép mũ hóa lặp lại. Với một số nguyên $i \geq 0$, biểu thức

$2^{2^{\cdot^{\cdot^{2}}}}\Big\}^{i}$

thay cho hàm $g(i)$, được định nghĩa theo đệ quy bởi

$$g(i) = \begin{cases} 2^1 & \text{nếu } i = 0\,, \\ 2^2 & \text{nếu } i = 1\,, \\ 2^{g(i-1)} & \text{nếu } i > 1\,. \end{cases}$$

Theo trực giác, tham số i cho "chiều cao của ngăn xếp gồm các 2" tạo thành số mũ. Ví dụ,

	$j = 1$	$j = 3$	$j = 3$	$j = 4$
$i = 1$	2^1	2^2	2^3	2^4
$i = 2$	2^2	2^{2^2}	$2^{2^{2^2}}$	$2^{2^{2^{2^2}}}$
$i = 3$	2^{2^2}	$2^{2^{\cdot^{\cdot^{2}}}}\Big\}16$	$2^{2^{\cdot^{\cdot^{2}}}}\Big\}2^{2^{\cdot^{\cdot^{2}}}}\Big\}16$	$2^{2^{\cdot^{\cdot^{2}}}}\Big\}2^{2^{\cdot^{\cdot^{2}}}}\Big\}2^{2^{\cdot^{\cdot^{2}}}}\Big\}16$

Hình 22.6 Các giá trị của $A(i, j)$ với các giá trị nhỏ của i và j.

$$2^{2^{\cdot^{\cdot^{2}}}}\Big\}4 = 2^{2^{2^{2}}} = 2^{65536}\,.$$

Hãy nhớ lại phần định nghĩa của hàm $\lg^*$ (trang 36) theo dạng các hàm $\lg^{(i)}$ được định nghĩa với số nguyên $i \geq 0$:

$$\lg^{(i)} n = \begin{cases} n & \text{nếu } i = 0\,, \\ \lg(\lg^{(i-1)} n) & \text{nếu } i > 0 \text{ và } \lg^{(i-1)} n > 0\,, \\ \text{không xác định} & \text{nếu } i > 0 \text{ và } \lg^{(i-1)} n \leq 0 \\ & \text{hoặc } \lg^{(i-1)} n \text{ là không xác định;} \end{cases}$$

$$\lg^* n = \min\{i \geq 0 : \lg^{(i)} n \leq 1\}\,.$$

Về cơ bản, hàm $\lg^*$ là nghịch đảo của phép mũ hóa lặp lại:

$$\lg^* 2^{2^{\cdot^{\cdot^{2}}}}\Big\}n = n + 1\,.$$

Giờ đây, ta đã sẵn sàng để nêu hàm Ackermann, được định nghĩa với các số nguyên $i, j \geq 1$ theo

$$A(1, j) = 2^j \qquad \text{với } j \geq 1,$$
$$A(i, 1) = A(i - 1, 2) \qquad \text{với } i \geq 2,$$
$$A(i, j) = A(i - 1, A(i, j - 1)) \qquad \text{với } i, j \geq 2.$$

Hình 22.6 nêu giá trị của hàm với các giá trị nhỏ của i và j.

Hình 22.7 nêu bằng sơ đồ tại sao hàm Ackermann có sự tăng trưởng bùng nổ như vậy. Hàng đầu tiên, hàm số mũ trong số cột j, đang tăng trưởng nhanh. Hàng thứ hai bao gồm tập hợp con chứa khoảng cách rộng gồm các cột $2, 2^2, 2^{2^2}, 2^{2^{2^2}}$,...của hàng đầu tiên. Các đường kẻ giữa các hàng kề nhau nêu rõ các cột trong hàng được đánh số thấp hơn nằm trong tập hợp con được gộp trong hàng đánh số cao hơn. Hàng thứ ba bao gồm tập hợp con thậm chí còn chứa khoảng cách rộng hơn gồm các cột $2, 2^{2^2}, 2^{2^{\cdot^{\cdot^{2}}}}\Big\}16$, $2^{2^{\cdot^{\cdot^{2}}}}\Big\}2^{2^{\cdot^{\cdot^{2}}}}\Big\}16$ .... của hàng thứ hai, là một tập hợp con thậm chí còn thưa hơn gồm các cột của hàng đầu tiên. Nói chung, khoảng cách giữa các cột của hàng i - 1 xuất hiện trong hàng i gia tăng đáng kể theo cả số cột lẫn số hàng. Nhận thấy $A(2, j) = 2^{2^{\cdot^{\cdot^{2}}}}\Big\}j$ với tất cả số nguyên $j \geq 1$. Như vậy, với $i > 2$, hàm $A(i, j)$ tăng trưởng thậm chí còn nhanh hơn $2^{2^{\cdot^{\cdot^{2}}}}\Big\}j$.

Hình 22.7 Sự tăng trưởng bùng nổ của hàm Ackermann. Các đường kẻ giữa các hàng i - 1 và i nêu rõ các khoản nhập của hàng i - 1 xuất hiện trong hàng i. Do sự tăng trưởng bùng nổ, khoảng cách ngang không theo tỷ lệ. Khoảng cách ngang giữa các khoản nhập của hàng i - 1 xuất hiện trong hàng i gia tăng mạnh theo số cột và số hàng. Nếu ta truy nguyên các khoản nhập trong hàng i theo dáng vẻ ban đầu của chúng trong hàng 1, sự tăng trưởng bùng nổ thậm chí còn rõ rệt hơn.

Ta định nghĩa nghịch đảo của hàm Ackermann theo[2]

$\alpha(m, n) = \min\{i \geq 1 : A(i, \lfloor m/n \rfloor) > \lg n\}$.

Nếu ta cố định một giá trị của n, thì khi m gia tăng, hàm $\alpha(m, n)$ giảm một cách đơn điệu. Để xem tính chất này, ta lưu ý $\lfloor m/n \rfloor$ tăng đơn điệu khi m gia tăng; do đó, bởi n cố định, nên giá trị nhỏ nhất của i cần có để mang $A(i, \lfloor m/n \rfloor)$ lên trên $\lg n$ sẽ giảm một cách đơn điệu. Tính chất này tương ứng với trực giác của chúng ta về các rừng tập hợp rời có heuristic nén lộ trình: với một số thành phần n riêng biệt đã cho, khi số lượng các phép toán m gia tăng, ta dự kiến chiều dài lộ trình tìm trung bình sẽ giảm do heuristic nén lộ trình. Nếu ta thực hiện m phép toán trong thời gian $O(m\ \alpha(m, n))$, thì thời gian trung bình của mỗi phép toán là $O(\alpha(m, n))$, sẽ giảm một cách đơn điệu khi m gia tăng.

Để trở về kiểu biện luận trên đây của chúng ta cho rằng $\alpha(m, n) \leq 4$ vì mọi mục tiêu thực tiễn, trước tiên ta lưu ý lượng $\lfloor m/n \rfloor$ ít nhất là 1, bởi $m \geq n$. Do hàm Ackermann tăng ngặt với mỗi đối số, $\lfloor m/n \rfloor \geq 1$ hàm ý $A(i, \lfloor m/n \rfloor) \geq A(i, 1)$ với tất cả $i \geq 1$. Nói cụ thể, $A(4, \lfloor m/n \rfloor) \geq A(4, 1)$. Song ta cũng có

$$\begin{aligned} A(4, 1) &= A(3, 2) \\ &= 2^{2^{\cdot^{\cdot^{2}}}}\Big\}16 \end{aligned}$$

lớn hơn nhiều so với số lượng nguyên tử dự trù trong vũ trụ quan sát được (xấp xỉ 10^{80}). Chỉ chình vì các giá trị lớn không thực tế của n mà $A(4, 1) << \lg n$, và như vậy $\alpha(m, n) \leq 4$ vì mọi mục tiêu thực tiễn. Lưu ý, cận $O(m \lg^* n)$ chỉ hơi yếu hơn cận $O(m\ \alpha(m, n))$; $\lg^* 65536 = 4$ và $\lg^* 2^{65536} = 5$, do đó $\lg^* n \leq 5$ vì mọi mục tiêu thực tiễn.

Các tính chất của các hạng

Trong phần còn lại của đoạn này, ta chứng minh một cận $O(m \lg^* n)$ trên thời gian thực hiện của các phép toán tập hợp rời với heuristic hợp theo hạng và nén lộ trình. Để chứng minh cận này, trước tiên ta chứng minh vài tính chất đơn giản của các hạng.

Bổ đề 22.2

Với tất cả các nút x, ta có $rank[x] \leq rank[p[x]]$, có bất đẳng thức ngặt

---

2 Tuy hàm này không phải là nghịch đảo của hàm Ackermann theo nghĩa toán học đích thực, song nó nắm giữ tinh thần của phép nghịch đảo trong sự tăng trưởng của nó, là càng chậm khi hàm Ackermann càng nhanh. Lý do mà ta dùng ngưỡng $\lg n$ thần bí sẽ được bộc lộ trong phần chứng minh của $O(m\ \alpha(m, n))$ thời gian thực hiện, vượt quá phạm vi của cuốn sách này.

nếu $x \neq p[x]$. Thoạt đầu giá trị của $rank[x]$ là 0 và gia tăng qua thời gian cho đến khi $x \neq p[x]$; từ đó trở đi, $rank[x]$ không thay đổi. Giá trị của $rank[p[x]]$ là một hàm thời gian tăng đơn điệu.

Chứng minh Phần chứng minh là một phương pháp quy nạp đơn giản trên số lượng các phép toán, dùng các thực thi của MAKE-SET, UNION, và FIND-SET xuất hiện trong Đoạn 22.3. Ta để nó lại làm Bài tập 22.4-1.

Ta định nghĩa size(x) là số lượng các nút trong cây có gốc tại nút x, kể cả chính nút x.

Bổ đề 22.3

Với tất cả các gốc cây x, size(x) ≥ $2^{rank[x]}$.

Chứng minh Ta chứng minh bằng phương pháp quy nạp trên số lượng các phép toán LINK. Lưu ý, các phép toán FIND-SET không làm thay đổi hạng của một cây gốc cũng như kích cỡ cây củanó.

Cơ sở: Bổ đề là đúng trước LINK đầu tiên, bởi các hạng thoạt đầu là 0 và mỗi cây chứa ít nhất một nút.

Bước quy nạp: Giả sử bổ đề là đúng trước khi thực hiện phép toán LINK(x, y). Cho *rank* thể hiện hạng ngay trước LINK, và cho *rank* thể hiện hạng ngay sau LINK. Định nghĩa size và size' tương tự.

Nếu $rank[x] \neq rank[y]$, mặc nhận mà không làm mất tính tổng quát rằng $rank[x] < rank[y]$. Nút y là gốc của cây được hình thành bởi phép toán LINK, và size'(y) = size(x) + size(y)

$$\begin{aligned} &\geq 2^{rank[x]} + 2^{rank[y]} \\ &\geq 2^{rank[y]} \\ &= 2^{rank'[y]} . \end{aligned}$$

Không có hạng hoặc kích cỡ nào thay đổi với bất kỳ nút nào khác ngoài y.

Nếu $rank[x] = rank[y]$, nút y lại là gốc của cây mới, và

$$\begin{aligned} \text{size}'(y) &= \text{size}(x) + \text{size}(y) \\ &\geq 2^{rank[x]} + 2^{rank[y]} \\ &= 2^{rank[y]+1} \\ &= 2^{rank'[y]} . \end{aligned}$$

Bổ đề 22.4

Với bất kỳ số nguyên $r \geq 0$, ta có tối đa $n/2^r$ nút có hạng r.

Chứng minh Cố định một giá trị cụ thể của r. Giả sử khi ta gán một

hạng r cho một nút x (trong dòng 2 của MAKE-SET hoặc trong dòng 5 của LINK), ta gắn một nhãn x với mỗi nút trong cây có gốc tại x. Qua Bổ đề 22.3, mỗi lần có ít nhất 2^r nút được gán nhãn. Giả sử gốc của cây chứa các thay đổi của nút x. Bổ đề 22.2 cam đoam với ta rằng hạng của gốc mới (hoặc, thực tế, của bất kỳ tiền bối riêng của x) ít nhất là $r+1$. Bởi ta chỉ gán các nhãn khi một gốc được gán một hạng r, không bao giờ nút trong cây mới này lại được gán nhãn một lần nữa. Như vậy, mỗi nút được gán nhãn tối đa một lần, khi lần đầu tiên gốc củanó được gán hạng r. Bởi có n nút, nên có tối đa n nút được gán nhãn, với ít nhất 2^r nhãn được gán cho mỗi nút có hạng r. Nếu có trên $n/2^r$ nút có hạng r, thì trên $2^r \bullet (n/2^r) = n$ nút sẽ được gán nhãn bởi một nút có hạng r, là một sự mâu thuẫn. Do đó, có tối đa $n/2^r$ nút được gán hạng r.

Hệ luận 22.5

Mọi nút có hạng tối đa $\lfloor \lg n \rfloor$.

Chứng minh Nếu ta cho $r > \lg n$, thì có tối đa $n/2^r < 1$ nút có hạng r. Bởi các hạng là các số tự nhiên, hệ luận sinh ra.

Chứng minh cận thời gian

Ta dùng phương pháp kết tập của phân tích khấu trừ (xem Đoạn 18.1) để chứng minh cận thời gian $O(m \lg^* n)$. Trong khi thực hiện phân tích khấu trừ, để tiện dụng, ta mặc nhận triệu gọi toán phép LINK thay vì phép toán UNION. Nghĩa là, do các tham số của tục thủ LINK là các biến trỏ đến hai gốc, nên ta mặc nhận rằng các phép toán FIND-SET thích hợp được thực hiện nếu cần. Bổ đề dưới đây chứng tỏ thậm chí nếu ta đếm các phép toán FIND-SET phụ trội, thời gian thực hiện tiệm cận vẫn giữ nguyên không đổi.

Bổ đề 22.6

Giả sử ta chuyển đổi một dãy S' gồm m' phép toán MAKE-SET, UNION, và FIND-SET thành một dãy S gồm m phép toán MAKE-SET, LINK, và FIND-SET bằng cách chuyển mỗi UNION thành hai phép toán FIND-SET theo sau là một LINK. Như vậy, nếu dãy S chạy trong $O(m \lg^* n)$ thời gian, dãy S' sẽ chạy trong $O(m' \lg^* n)$ thời gian.

Chứng minh Do mỗi phép toán UNION trong dãy S' được chuyển đổi thành ba phép toán trong S, ta có $m' \le m \le 3m'$. Bởi $m = O(m')$, một cận thời gian $O(m \lg^* n)$ với dãy S đã chuyển đổi hàm ý một cận thời gian $O(m' \lg^* n)$ với dãy S' ban đầu.

Trong phần còn lại của đoạn này, ta mặc nhận dãy ban đầu gồm m'

phép toán MAKE-SET, UNION, và FIND-SET đã được chuyển đổi thành một một dãy m phép toán MAKE-SET, LINK, và FIND-SET. Giờ đây, ta chứng minh một cận thời gian $O(m \lg^* n)$ cho dãy được chuyển đổi và viện đến Bổ đề 22.6 để chứng minh thời gian thực hiện $O(m' \lg^* n)$ của dãy ban đầu của m' phép toán.

Định lý 22.7

Một dãy m phép toán MAKE-SET, LINK, và FIND-SET, n của chúng là các phép toán MAKE-SET, có thể được thực hiện trên một rừng tập hợp rời bằng hợp theo hạng và nén lộ trình trong thời gian trường hợp xấu nhất $O(m \lg^* n)$.

Chứng minh Ta ước định các ***khoản tính công*** tương ứng với mức hao phí thực tế của mỗi phép toán tập hợp và tính tổng các khoản tính công đã ước tính một khi nguyên cả dãy các phép toán tập hợp đã được thực hiện. Như vậy, tổng này cho ta mức hao phí thực tế của tất cả các phép toán tập hợp.

Các khoản tính công đã ước tính cho các phép toán MAKE-SET và LINK chẳng có gì phức tạp: một khoản tính công cho mỗi phép toán. Bởi từng phép toán này mất $O(1)$ thời gian thực tế, nên các khoản tính công đã ước tính sẽ bằng với các mức hao phí thực tế của các phép toán.

Trước khi đề cập các khoản tính công ước tình cho các phép toán FIND-SET, ta phân hoạch các hạng nút thành các ***khối*** [blocks] bằng cách đặt hạng r vào khối $\lg^* r$ với $r = 0, 1, ..., \lfloor \lg n \rfloor$. (Hãy nhớ lại $\lfloor \lg n \rfloor$ là hạng cực đại.) Do đó, khối được đánh số cao nhất là khối $\lg^* (\lg n) = \lg^* n - 1$. Để tiện dụng về ký hiệu, ta định nghĩa với các số nguyên $j \geq -1$,

$$B(j) = \begin{cases} -1 & \text{nếu } j = -1 , \\ 1 & \text{nếu } j = 0 , \\ 2 & \text{nếu } j = 1 , \\ 2^{2^{\cdot^{\cdot^{2}}}}\Big\}^{j-1} & \text{nếu } j \geq 2 . \end{cases}$$

Như vậy, với $j = 0, 1, ..., \lg^* n - 1$, khối thứ j bao gồm tập hợp các hạng

$\{B(j - 1) + 1, B(j - 1) + 2, ... , B(j)\}$.

Ta dùng hai kiểu khoản tính công cho một phép toán FIND-SET: ***các khoản tính công khối*** [block charges] và ***các khoản tính công lộ trình***

[path charges]. Giả sử FIND-SET bắt đầu tại nút x_0 và lộ trình tìm bao gồm các nút $x_0, x_1, ..., x_l$, ở đó với $i = 1, 2, ..., l$, nút x_i là $p[x_{i-1}]$ và x_l (một gốc) là $p[x_l]$. Với $j = 0, 1, ..., \lg^* n - 1$, ta ước định một khoản tính công khối cho nút *last* với hạng trong khối *j* trên lộ trình. (Lưu ý, Bổ đề 22.2 hàm ý rằng trên một lộ trình tìm bất kỳ, các nút có các hạng trong một khối đã cho sẽ là liên tục.) Ta cũng ước định một khoản tính công khối cho con của gốc, nghĩa là, cho x_{l-1}. Do các hạng gia tăng ngặt dọc theo một lộ trình tìm bất kỳ, nên một kiểu trình bày tương đương sẽ ước định một khoản tính công khối cho mỗi nút x_i sao cho $p[x_i] = x_l$ (x_i là gốc hoặc con của nó) hoặc $\lg^* rank[x_i] < \lg^* rank[x_{i+1}]$ (khối của hạng của x_i khác với của cha của nó). Tại mỗi nút trên lộ trình tìm mà ta không ước định một khoản tính công khối, ta ước định một khoản tính công lộ trình.

Sau khi một nút ngoài gốc hoặc con của nó được ước tính các khoản tính công khối, nó sẽ không bao giờ được ước tính lại theo các khoản tính công lộ trình. Để xem tại sao, ta nhận thấy mỗi lần phương pháp nén lộ trình xảy ra, hạng của một nút x_i mà $p[x_i] \neq x_l$ giữ nguyên không đổi, song cha mới của x_i có một hạng lớn ngặt hơn của cha cũ của x_i. Sự khác biệt giữa các hạng của x_i và cha của nó là một hàm thời gian tăng đơn điệu. Như vậy, sự khác biệt giữa $\lg^* rank[p[x_i]]$ và $\lg^* rank[x_i]$ cũng là một hàm thời gian tăng đơn điệu. Một khi x_i và cha của nó có các hạng trong các khối khác nhau, chúng sẽ luôn có các hạng trong các khối khác nhau, và do đó x_i sẽ chẳng bao giờ được ước tính lại một khoản tính công lộ trình.

Bởi ta đã tính công một lần cho mỗi nút ghé thăm trong mỗi phép toán FIND-SET, nên tổng các khoản tính công ước tính sẽ là tổng các nút ghé thăm trong tất cả các phép toán FIND-SET; tổng này biểu diễn mức hao phí thực tế của tất cả các phép toán FIND-SET. Ta muốn chứng tỏ tổng này là $O(m \lg^* n)$.

Ta có thể dễ dàng định cận số lượng các khoản tính công khối. Có tối đa một khoản tính công khối được ước tính cho mỗi số khối trên lộ trình tìm đã cho, cộng với một khoản tính công khối cho con của gốc. Bởi các số khối nằm trong phạm vi từ 0 đến $\lg^* n - 1$, nên ta có tối đa $\lg^* n + 1$ khoản tính công khối được ước tính cho mỗi phép toán FIND-SET. Như vậy, ta có tối đa $m(\lg^* n + 1)$ khoản tính công khối ước tính trên tất cả các phép toán FIND-SET.

Việc định cận các khoản tính công lộ trình có hơi tinh tế hơn. Ta bắt đầu bằng cách nhận xét rằng nếu một nút x_i được ước tính một khoản

tính công lộ trình, thì $p[x_i] \neq x_i$ trước khi nén lộ trình, sao cho x_i sẽ được gán một cha mới trong khi nén lộ trình. Hơn nữa, như ta đã nhận thấy, cha mới x_i có một hạng cao hơn cha cũ của nó. Giả sử hạng của nút x_i nằm trong khối j. Có thể gán x_i cho một cha mới bao nhiêu lần, và do đó được ước tính một khoản tính công lộ trình, trước khi x_i được gán một cha có hạng nằm trong một khối khác (mà sau đó x_i sẽ không bao giờ được ước tính lại một khoản tính công lộ trình)? Số lần này là cực đại nếu x_i có hạng thấp nhất trong khối của nó, tức $B(j-1)+1$, và các hạng của các cha của nó liên tục tiếp nhận các giá trị $B(j-1)+2$, $B(j-1)+3, \ldots, B(j)$.

Do có $B(j) - B(j-1) - 1$ hạng như vậy, nên ta kết luận rằng một đỉnh có thể được ước tính tối đa $B(j) - B(j-1) - 1$ khoản tính công lộ trình trong khi hạng của nó nằm trong khối j.

Bước kế tiếp của chúng ta trong tiến trình định cận các khoản tính công lộ trình đó là định cận số lượng các nút có các hạng trong khối j với các số nguyên $j \geq 0$. (Hãy nhớ lại qua Bổ đề 22.2, hạng của một nút được cố định một khi nó trở thành một con của một nút khác.) Cho số lượng nút có các hạng nằm trong khối j được thể hiện bởi $N(j)$. Thì, qua Bổ đề 22.4,

$$N(j) \leq \sum_{r=B(j-1)+1}^{B(j)} \frac{n}{2^r} .$$

Với $j = 0$, tổng này đánh giá là

$$\begin{aligned} N(0) &= n/2^0 + n/2^1 \\ &= 3n/2 \\ &= 3n/2B(0) . \end{aligned}$$

Với $j \geq 1$, ta có

$$\begin{aligned} N(j) &\leq \frac{n}{2^{B(j-1)+1}} \sum_{r=0}^{B(j)-(B(j-1)+1)} \frac{1}{2^r} . \\ &< \frac{n}{2^{B(j-1)+1}} \sum_{r=0}^{\infty} \frac{1}{2^r} \\ &= \frac{n}{2^{B(j-1)}} \\ &= \frac{n}{B(j)} . \end{aligned}$$

Như vậy, $N(j) \leq 3n/2B(j)$ với tất cả các số nguyên $j \geq 0$.

Ta hoàn tất việc định cận các khoản tính công lộ trình bằng cách tính tổng trên tất cả các khối tích của số lượng cực đại các nút có các hạng trong khối và số lượng cực đại các khoản tính công lộ trình cho mỗi nút của khối đó. Thể hiện bằng $P(n)$ số lượng chung các khoản tính công lộ trình, ta có

$$\begin{aligned}
P(n) &\le \sum_{j=0}^{\lg^* n - 1} \frac{3n}{2B(j)} (B(j) - B(j-1) - 1) \\
&\le \sum_{j=0}^{\lg^* n - 1} \frac{3n}{2B(j)} \cdot B(j) \\
&= \frac{3}{2} n \lg^* n \,.
\end{aligned}$$

Như vậy, tổng các khoản tính công mà các phép toán FIND-SET gánh chịu là $O(m(\lg^* n + 1) + n \lg^* n)$, tức là $O(m \lg^* n)$ bởi $m \ge n$. Do ta có $O(n)$ phép toán MAKE-SET và LINK, với một khoản tính công cho mỗi phép toán, tổng thời gian là $O(m \lg^* n)$.

Hệ luận 22.8

Một dãy m phép toán MAKE-SET, UNION, và FIND-SET, n trong đó là các phép toán MAKE-SET, có thể được thực hiện trên một rừng tập hợp rời bằng phương pháp hợp theo hạng và nén lộ trình trong thời gian ca xấu nhất $O(m \lg^* n)$.

Chứng minh Tức thời từ Định lý 22.7 và Bổ đề 22.6.

Bài tập

22.4-1

Chứng minh Bổ đề 22.2.

22.4-2

Với mỗi nút x, ta cần bao nhiêu bit để lưu trữ size(x)? $rank[x]$ thì sao?

22.4-3

Dùng Bổ đề 22.2 và Hệ luận 22.5, cho một chứng minh đơn giản rằng các phép toán trên một rừng tập hợp rời bằng heuristic hợp theo hạng nhưng không có nén lộ trình chạy trong $O(m \lg n)$ thời gian.

22.4-4 *

Giả sử ta sửa đổi quy tắc về cách ước định các khoản tính công để ta

ước định một khoản tính công khối cho nút cuối trên lộ trình tìm có hạng nằm trong khối j với $j = 0, 1, \ldots, \lg^* n - 1$. Bằng không, ta ước định một khoản tính công lộ trình cho nút. Như vậy, nếu một nút là một con của gốc và không phải là nút cuối của một khối, nó được ước tính một khoản tính công lộ trình, chứ không phải một khoản tính công khối. Chứng tỏ $\Omega(m)$ khoản tính công lộ trình có thể được ước tính cho một nút đã cho trong khi hạng của nó nằm trong một khối j đã cho.

Các Bài Toán

22-1 Cực tiểu ngoài tuyến

Bài toán cực tiểu ngoài tuyến yêu cầu ta duy trì một tập hợp động T gồm các thành phần từ miền xác định $\{1, 2, \ldots, n\}$ dưới các phép toán INSERT và EXTRACT-MIN. Được biết một dãy S n lệnh gọi INSERT và m lệnh gọi EXTRACTMIN, ở đó mỗi khóa trong $\{1, 2, \ldots, n\}$ được chèn chính xác một lần. Ta muốn xác định khóa được mỗi lệnh gọi EXTRACT-MIN trả về. Cụ thể, ta muốn điền một mảng *extracted*[1. . *m*], ở đó với $i = 1, 2, \ldots, m$, *extracted*[*i*] là khóa mà lệnh gọi EXTRACT-MIN thứ i trả về. Bài toán là "ngoài tuyến" theo nghĩa là ta được phép xử lý nguyên cả dãy S trước khi xác định bất kỳ khóa nào được trả về.

a. Trong trường hợp dưới đây của bài toán cực tiểu ngoài tuyến, mỗi INSERT được biểu thị bằng một số và mỗi EXTRACT-MIN được biểu thị bằng mẫu tự E:

4, 8, E, 3, E, 9, 2, 6, E, E, E, 1, 7, E, 5 .

Điền các giá trị đúng đắn trong mảng *extracted*.

Để phát triển một thuật toán cho bài toán này, ta tách dãy S thành các dãy con thuần chủng. Nghĩa là, ta biểu diễn S theo

I_1, E, I_2, E, I_3, ..., I_m, E, I_{m+1} ,

ở đó mỗi E biểu diễn một lệnh gọi EXTRACT-MIN đơn lẻ và mỗi I_j biểu diễn một dãy (có thể trống) các lệnh INSERT. Với mỗi dãy con I_j, thoạt đầu ta đưa các khóa mà các phép toán này chèn vào một tập hợp K_j, là trống nếu I_j là trống. Sau đó, ta thực hiện như sau.

OFF-LINE-MINIMUM(m, n)

1 **for** $i \leftarrow 1$ **to** n

2 **do** xác định j sao cho $i \in K_j$

```
        if j ≠ m + 1
           then extracted [ j ] ← i
                  cho l là giá trị nhỏ nhất lớn hơn j
                     mà tập hợp K_l tồn tại
                  K_l ← K_j ∪ K_l, hủy K_j
return extracted
```

b. Chứng tỏ mảng *extracted* mà OFF-LINE-MINIMUM trả về là đúng đắn.

c. Mô tả cách sử dụng một cấu trúc dữ liệu tập hợp rời để thực thi OFF-LINE-MINIMUM một cách hiệu quả. Nêu một cận chặt trên thời gian thực hiện ca xấu nhất của kiểu thực thi của bạn.

22-2 Xác định độ sâu

Trong ***bài toán xác định độ sâu***, ta duy trì một rừng $\mathcal{F} = \{T_i\}$ gồm các cây có gốc dưới ba phép toán:

MAKE-TREE(v) tạo một cây có nút duy nhất là v.

FIND-DEPTH(v) trả về độ sâu của nút v trong cây của nó.

GRAFT(r, v) tạo nút r, được mặc nhận là gốc của một cây, trở thành con của nút v, được mặc nhận nằm trong một cây khác với r nhưng có thể hoặc không là một gốc.

a. Giả sử ta dùng một phép biểu diễn cây tương tự như một rừng tập hợp rời: $p[v]$ là cha của nút v, ngoại trừ $p[v] = v$ nếu v là một gốc. Nếu ta thực thi GRAFT(r, v) bằng cách ấn định $p[r] \leftarrow v$ và FIND-DEPTH(v) bằng cách theo lộ trình tìm lên đến gốc, trả về một số đếm tất cả các nút khác ngoài v đã gặp, chứng tỏ thời gian thực hiện ca xấu nhất của một dãy m phép toán MAKE-TREE, FIND-DEPTH, và GRAFT là $\Theta(m^2)$.

Nhờ dùng các heuristic hợp theo hạng và nén lộ trình, ta có thể rút gọn thời gian thực hiện ca xấu nhất. Ta dùng rừng tập hợp rời $S = \{S_i\}$, ở đó mỗi tập hợp S_i (chính là một cây) tương ứng với một cây T_i trong rừng $\mathcal{F}$. Tuy nhiên, cây cấu trúc trong một tập hợp S_i không nhất thiết tương ứng với của T_i. Thực vậy, cách thực thi của S_i không ghi nhận các mối quan hệ cha-con chính xác nhưng tuy vậy cho phép ta xác định độ sâu của một nút bất kỳ trong T_i.

Ý tưởng chính đó là duy trì trong mỗi nút v một "khoảng cách giả" $d[v]$, được định nghĩa sao cho tổng các khoảng cách giả dọc theo lộ trình

từ v đến gốc của tập hợp S_i của nó bằng độ sâu của v trong T_i. Nghĩa là, nếu lộ trình từ v các gốc của nó trong S_i là $v_0, v_1, ..., v_k$, ở đó $v_0 = v$ và v_k là gốc của S_i, thì độ sâu của v trong T_i là $\sum_{j=0}^{k} d[v_j]$.

b. Nêu một cách thực thi của MAKE-TREE.

c. Nêu cách sửa đổi FIND-SET để thực thi FIND-DEPTH. Cách thực thi của bạn phải thực hiện heuristic nén lộ trình, và thời gian thực hiện củanó phải tuyến tính theo chiều dài của lộ trình tìm. Bảo đảm cách thực thi của bạn cập nhật đúng đắn các khoảng cách giả.

d. Nêu cách sửa đổi các thủ tục UNION và LINK để thực thi GRAFT(r, v), tổ hợp các tập hợp chứa r và v. Bảo đảm cách thực thi của bạn cập nhật các khoảng cách giả đúng đắn. Lưu ý, gốc của một tập hợp S_i không nhất thiết là gốc của cây T_i tương ứng.

e. Nêu một cận chặt trên thời gian thực hiện ca xấu nhất của một dãy m phép toán MAKE-TREE, FIND-DEPTH, và GRAFT, n trong số đó là các phép toán MAKE-TREE.

22-3 Thuật toán các tiền bối chung nhỏ nhất ngoài tuyến của Tarjan

Tiền bối chung nhỏ nhất của hai nút u và v trên một cây có gốc T là nút w và là một tiền bối của cả u lẫn v và có độ sâu lớn nhất trong T. Trong ***bài toán các tiền bối chung nhỏ nhất ngoài tuyến,*** ta có một cây T có gốc và một tập hợp tùy ý $P = \{\{u, v\}\}$ của các cặp không có thứ tự gồm các nút trong T, và ta muốn xác định tiền bối chung nhỏ nhất của mỗi cặp trong P.

Để giải quyết bài toán các tiền bối chung nhỏ nhất ngoài tuyến, thủ tục dưới đây thực hiện một tầng cây của T với lệnh gọi ban đầu LCA(*root*[T]). Mỗi nút được mặc nhận tô màu WHITE trước bước duyệt [walk].

```
LCA(u)
MAKE-SET(u)
ancestor[FIND-SET(u)] ← u
for mỗi con v của u trong T
    do LCA(v)
       UNION(u, v)
       ancestor[FIND-SET(u)] ← u
color[u] ← BLACK
```

```
8   for mỗi nút v sao cho {u, v} ∈ P
9       do if color[v] = BLACK
10          then print "Tiền bối chung nhỏ nhất của"
                    u "và" v "là" ancestor[FIND-SET(v)]
```

a. Chứng tỏ dòng 10 được thi hành chính xác một lần cho mỗi cặp $\{u,v\} \in P$.

b. Chứng tỏ vào lúc gọi LCA(u), số lượng tập hợp trong cấu trúc dữ liệu tập hợp rời bằng với độ sâu của u trong T.

c. Chứng minh LCA in đúng đắn tiền bối chung nhỏ nhất của u và v với từng cặp $\{u, v\} \in P$.

d. Phân tích thời gian thực hiện của LCA, giả định ta dùng cách thực thi của cấu trúc dữ liệu tập hợp rời trong Đoạn 22.3.

Ghi chú Chương

Nhiều kết quả quan trọng về các cấu trúc dữ liệu tập hợp rời chí ít phần nào là do R. E. Tarjan. Tarjan [186, 188] là người đầu tiên cung cấp cận trên của $O(m\,\alpha(m, n))$. $O(m \lg^* n)$ cận trên đã được Hopcroft và Ullman [4, 103] chứng minh. Tarjan và van Leeuwen [190] đề cập các các biến thức trên heuristic nén lộ trình, kể cả "các phương pháp một lượt," đôi lúc cung cấp các thừa số bất biến về khả năng thực hiện tốt hơn các phương pháp hai lượt. Gabow và Tarjan [76] chứng tỏ trong một số ứng dụng nhất định, có thể thực hiện các phép toán tập hợp rời để chạy trong $O(m)$ thời gian.

Tarjan [187] đã chứng tỏ cần có một cận dưới của $\Omega(m\alpha(m, n))$ thời gian cho các phép toán trên mọi cấu trúc dữ liệu tập hợp rời thỏa một số điều kiện kỹ thuật nhất định. Cận dưới này về sau đã được tổng quát hóa bởi Fredman và Saks [74], họ đã chứng tỏ trong trường hợp xấu nhất, $\Omega(m\,\alpha(m, n))$ từ [word] ($\lg n$) bit của bộ nhớ phải được truy cập.

VI Thuật Toán Đồ Thị

Mở đầu

Đồ thị là một cấu trúc dữ liệu phổ biến trong khoa học máy tính, và các thuật toán để làm việc với chúng là cần thiết cho lĩnh vực này. Có hàng trăm bài toán điện toán đáng quan tâm được định nghĩa theo dạng đồ thị. Trong phần này, ta đề cập một số bài toán quan trọng.

Chương 23 nêu cách biểu diễn một đồ thị trên một máy tính rồi mô tả các thuật toán dựa trên việc tìm kiếm trong một đồ thị bằng phương pháp tìm kiếm độ rộng đầu tiên hoặc tìm kiếm độ sâu đầu tiên. Ta sẽ mô tả hai ứng dụng của phương pháp tìm kiếm độ sâu đầu tiên: sắp xếp theo tôpô một đồ thị phi chu trình có hướng và phân tích một đồ thị có hướng thành các thành phần liên thông mạnh của nó.

Chương 24 mô tả cách tính một cây tỏa nhánh có trọng số cực tiểu của một đồ thị. Một cây như vậy được định nghĩa là con đường có trọng số nhỏ nhất để liên thông tất cả các đỉnh với nhau khi mỗi cạnh có một trọng số kết hợp. Các thuật toán để tính toán các cây tỏa nhánh cực tiểu là những ví dụ tốt về các thuật toán tham (xem Chương 17).

Các chương 25 và 26 sẽ xét bài toán tính các lộ trình ngắn nhất giữa các đỉnh khi mỗi cạnh có một chiều dài hoặc "trọng số" kết hợp. Chương 25 xem xét cách tính các lộ trình ngắn nhất từ một đỉnh nguồn đã cho đến tất cả các đỉnh khác, và Chương 26 giải thích cách tính các lộ trình ngắn nhất giữa mọi cặp đỉnh.

Cuối cùng, Chương 27 nêu cách tính toán một luồng vật liệu cực đại trong một mạng (đồ thị có hướng) có một nguồn vật liệu đãđịnh, một bồn đãđịnh, và các dung lượng đã định cho lượng vật liệu có thể băng ngang mỗi cạnh co hướng. Bài toán chung này nảy sinh theo nhiều dạng, và có thể dung một thuật toán tốt để tính toán các luồng cực đại nhằm giải quyết hiệu quả nhiều bài toán có liên quan.

Trong khi mô tả thời gian thực hiện của một thuật toán đồ thị trên một đồ thị đã cho $G = (V, E)$, ta thường đo kích cỡ nhập liệu theo dạng số lượng các đỉnh $|V|$ và số lượng các cạnh $|E|$ của đồ thị. Nghĩa là, có

hai tham số liên quan mô tả kích cỡ nhập liệu, chứ không chỉ một. Ta chấp nhận một quy ước ký hiệu chung cho các tham số này. Bên trong hệ ký hiệu tiệm cận (như hệ ký hiệu O hoặc Θ), và *chỉ* bên trong hệ ký hiệu đó, ký hiệu V mới thể hiện $|V|$ và ký hiệu E thể hiện $|E|$. Ví dụ, ta có thể nói, "thuật toán chạy trong thời gian $O(V E)$," nghĩa là thuật toán chạy trong thời gian $O(|V| |E|)$. Quy ước này khiến các công thức thời gian thực hiện dễ đọc hơn, mà không bị tối nghĩa.

Một quy ước khác mà ta chấp nhận xuất hiện trong mã giả. Ta thể hiện tập hợp đỉnh của một đồ thị G bằng $V[G]$ và tập hợp cạnh của nó bằng $E[G]$. Nghĩa là, mã giả xem các tập hợp đỉnh và cạnh như là các thuộc tính của một đồ thị.

23 Các Thuật Toán Đồ Thị Căn Bản

Chương này trình bày các phương pháp để biểu thị một đồ thị và để tìm kiếm trong đồ thị. Tìm kiếm trong một đồ thị có nghĩa là theo các cạnh đồ thị một cách có hệ thống để ghé thăm các đỉnh của đồ thị. Một thuật toán tìm kiếm đồ thị có thể khám phá nhiều về cấu trúc của một đồ thị. Nhiều thuật toán bắt đầu bằng cách tìm kiếm trong đồ thị nhập liệu của chúng để có được thông tin cấu trúc này.

Các thuật toán đồ thị khác được tổ chức dưới dạng cụ thể hóa đơn giản các thuật toán tìm kiếm đồ thị cơ bản. Các kỹ thuật để tìm kiếm trong một đồ thị là trung tâm của lĩnh vực các thuật toán đồ thị.

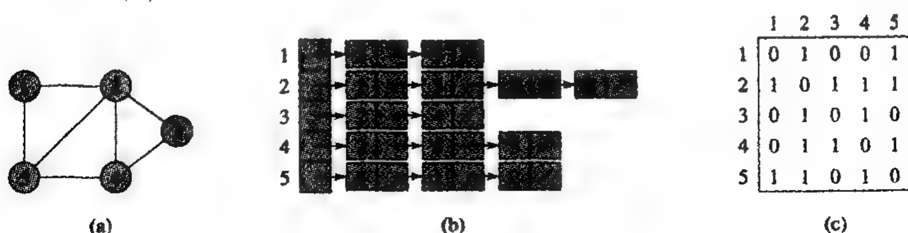
Đoạn 23.1 mô tả hai phép biểu diễn tính toán thông dụng nhất của đồ thị: dưới dạng các danh sách kề và các ma trận kề. Đoạn 23.2 trình bày một thuật toán tìm kiếm đồ thị đơn giản có tên tìm kiếm độ rộng đầu tiên và nêu cách tạo một cây độ rộng đầu tiên. Đoạn 23.3 trình bày thuật toán tìm kiếm độ sâu đầu tiên và chứng minh vài kết quả chuẩn về thứ tự ở đó thuật toán tìm kiếm độ sâu đầu tiên ghé thăm các đỉnh. Đoạn 23.4 cung cấp ứng dụng thực đầu tiên về tìm kiếm độ sâu đầu tiên: sắp xếp theo tôpô một đồ thị phi chu trình có hướng. Đoạn 23.5 nêu một ứng dụng thứ hai về tìm kiếm độ sâu đầu tiên: tìm các thành phần liên thông mạnh của một đồ thị có hướng.

23.1 Các phép biểu diễn của đồ thị

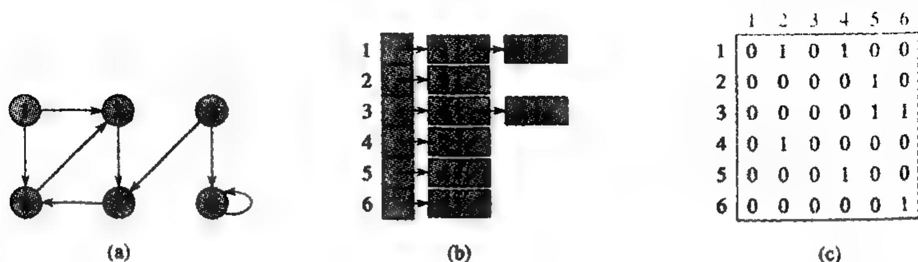
Có hai cách chuẩn để biểu diễn một đồ thị $G = (V, E)$: dưới dạng một tập hợp các danh sách kề hoặc một ma trận kề. Phép biểu diễn danh sách kề thường được ưa dùng, bởi nó cung cấp một cách nén gọn để biểu diễn các đồ thị ***thưa*** [sparse graph]—là những đồ thị mà $|E|$ nhỏ hơn nhiều so với $|V|^2$. Hầu hết các thuật toán đồ thị trình bày trong cuốn sách này mặc nhận một đồ thị nhập liệu được biểu thị theo dạng danh sách kề. Tuy nhiên, phép biểu diễn ma trận kề có thể được ưa dùng khi đồ thị là ***trù mật*** [dense]—$|E|$ sát với $|V|^2$—hoặc khi ta cần nhanh chóng biết có một cạnh nối hai đỉnh đã cho hay không. Ví dụ, hai trong số các thuật toán các lộ trình ngắn nhất với tất cả các cặp được

trình bày trong Chương 26 mặc nhận rằng đồ thị nhập liệu của chúng được biểu thị bởi các ma trận kề.

Phép biểu diễn danh sách kề [adjacency-list representation] của một đồ thị $G = (V, E)$ bao gồm một mảng *Adj* gồm $|V|$ danh sách, một cho từng đỉnh trong V. Với từng $u \in V$, danh sách kề *Adj*[u] chứa (trỏ đến) tất cả các đỉnh v sao cho có một cạnh $(u, v) \in E$. Nghĩa là, *Adj*[u] bao gồm tất cả các đỉnh kề với u trong G. Các đỉnh trong mỗi danh sách kề thường được lưu trữ theo thứ tự tùy ý. Hình 23.1(b) là một phép biểu diễn danh sách kề của đồ thị không có hướng trong Hình 23.1(a). Cũng vậy, Hình 23.2(b) là một phép biểu diễn danh sách kề của đồ thị có hướng trong Hình 23.2(a).



Hình 23.1 Hai phép biểu diễn của một đồ thị không có hướng. **(a)** Một đồ thị không có hướng G có năm đỉnh và bảy cạnh. **(b)** Một phép biểu diễn danh sách kề của G. **(c)** Phép biểu diễn ma trận kề của G.



Hình 23.2 Hai phép biểu diễn của một đồ thị có hướng. **(a)** Một đồ thị có hướng G có sáu đỉnh và tám cạnh. **(b)** Một phép biểu diễn danh sách kề của G. **(c)** Phép biểu diễn ma trận kề của G.

Nếu G là một đồ thị có hướng, tổng các chiều dài của tất cả các danh sách kề là $|E|$, bởi một cạnh của dạng (u, v) được biểu thị bằng cách để v xuất hiện trong *Adj*[u]. Nếu G là một đồ thị không có hướng, tổng các chiều dài của tất cả các danh sách kề là $2|E|$, bởi nếu (u, v) là một cạnh không có hướng, thì u xuất hiện trong danh sách kề của v và ngược lại. Dẫu một đồ thị có hướng hay không, phép biểu diễn danh sách kề có tính chất thỏa đáng đó là lượng bộ nhớ mà nó yêu cầu là $O(\max(V, E)) = O(V + E)$.

Các danh sách kề có thể sẵn sàng được thích ứng để biểu diễn ***đồ thị***

trọng số {weighted graphs], nghĩa là, đồ thị mà mỗi cạnh có một ***trọng số*** kết hợp, thường được cung cấp bởi một ***hàm trọng số*** $w : E \to \mathbf{R}$. Ví dụ, cho $G = (V, E)$ là một đồ thị gia trọng với hàm trọng số w. Trọng số $w(u,v)$ của cạnh $(u,v) \in E$ đơn giản được lưu trữ với đỉnh v trong danh sách kề của u. Phép biểu diễn danh sách kề tỏ ra khá cường tráng ở chỗ nó có thể được sửa đổi để hỗ trợ nhiều biến thức đồ thị khác.

Một nhược điểm tiềm ẩn của phép biểu diễn danh sách kề đó là để xác định xem một cạnh đã cho (u, v) có hiện diện trong đồ thị hay không, ta không có cách nào nhanh hơn là tìm kiếm v trong danh sách kề $Adj[u]$. Có thể khắc phục khuyết điểm này bằng một phép biểu diễn ma trận kề của đồ thị, để đổi lại ta phải dùng nhiều bộ nhớ hơn theo tiệm cận.

Với ***phép biểu diễn ma trận kề*** của một đồ thị $G = (V, E)$, ta mặc nhận các đỉnh được đánh số 1, 2,..., $|V|$ theo một kiểu tùy ý. Như vậy, phép biểu diễn ma trận kề của một đồ thị G bao gồm một ma trận $|V| \times |V|$ $A = (a_{ij})$ sao cho

$$a_{ij} = \begin{cases} 1 \text{ nếu } (i, j) \in E, \\ 0 \text{ bằng không .} \end{cases}$$

Các Hình 23.1(c) và 23.2(c) là các ma trận kề của đồ thị không có hướng và có hướng trong các Hình 23.1(a) và 23.2(a), theo thứ tự nêu trên. Ma trận kề của một đồ thị yêu cầu $\Theta(V^2)$ bộ nhớ, độc lập với số lượng các cạnh trong đồ thị.

Lưu ý tính đối xứng dọc theo đường chéo chính của ma trận kề trong Hình 23.1(c). Ta định nghĩa ***hoán vị*** của một ma trận $A = (a_{ij})$ là ma trận $A^T = (a^T_{ij})$ với $a^T_{ij} = a_{ji}$. Bởi trong một đồ thị không có hướng, (u, v) và (v, u) biểu diễn cùng cạnh, ma trận kề A của một đồ thị không có hướng là chuyển vị riêng củanó: $A = A^T$. Trong vài ứng dụng, ta sẽ có lợi khi chỉ lưu trữ các khoản nhập trên và bên trên đường chéo của ma trận kề, nhờ đó cắt giảm hầu như một nửa lượng bộ nhớ cần có để lưu trữ đồ thị.

Cũng như phép biểu diễn danh sách kề của một đồ thị, phép biểu diễn ma trận kề có thể được dùng cho các đồ thị gia trọng. Ví dụ, nếu $G = (V, E)$ là một đồ thị gia trọng có hàm trọng số cạnh w, trọng số $w(u, v)$ của cạnh $(u, v) \in E$ đơn giản được lưu trữ dưới dạng khoản nhập trong hàng u và cột v của ma trận kề. Nếu một cạnh không tồn tại, một giá trị NIL có thể được lưu trữ dưới dạng khoản nhập ma trận tương ứng của-nó, dù với nhiều bài toán việc sử dụng một giá trị như 0 hoặc ∞ sẽ tiện dụng hơn.

Mặc dù theo tiệm cận, phép biểu diễn danh sách kề ít nhất cũng hiệu quả như phép biểu diễn ma trận kề, song tính đơn giản của một ma

trận kề có thể khiến nó được ưa dùng khi đồ thị tương đối nhỏ. Hơn nữa, nếu đồ thị không gia trọng, phép biểu diễn ma trận kề còn có thêm một ưu điểm về kho lưu trữ. Thay vì dùng một từ của bộ nhớ máy tính cho từng khoản nhập ma trận, ma trận kề chỉ sử dụng một bit cho mỗi khoản nhập.

Bài tập

23.1-1

Cho một phép biểu diễn danh sách kề của một đồ thị có hướng, ta phải mất bao lâu để tính toán độ-ra [out-degree] của mọi đỉnh? Phải mất bao lâu để tính toán các độ-vào [in-degrees]?

23.1-2

Nêu một phép biểu diễn danh sách kề cho một cây nhị phân hoàn chỉnh trên 7 đỉnh. Nêu một phép biểu diễn ma trận kề tương đương. Giả sử các đỉnh được đánh số từ 1 đến 7 như trong một đống nhị phân.

23.1-3

Hoán vị của một đồ thị có hướng $G = (V, E)$ là đồ thị $G^T = (V, E^T)$, ở đó $E^T = \{(v, u) \in V \times V : (u, v) \in E\}$. Như vậy, G^T là G với mọi cạnh của nó được đảo nghịch. Mô tả hiệu quả các thuật toán để tính toán G^T từ G, với cả phép biểu diễn danh sách kề lẫn ma trận kề của G. Phân tích thời gian thực hiện của các thuật toán.

23.1-4

Cho một phép biểu diễn danh sách kề của một đa đồ thị $G = (V, E)$, mô tả một thuật toán $O(V + E)$ thời gian để tính toán phép biểu diễn danh sách kề của đồ thị không có hướng "tương đương" $G' = (V, E')$, ở đó E' bao gồm các cạnh trong E với tất cả đa cạnh giữa hai đỉnh được thay bằng một cạnh đơn và với tất cả các vòng tự lặp được gỡ bỏ.

23.1-5

Bình phương của một đồ thị có hướng $G = (V, E)$ là đồ thị $G^2 = (V, E^2)$ sao cho $(u, w) \in E^2$ nếu và chỉ nếu với một $v \in V$, cả hai $(u, v) \in E$ và $(v, w) \in E$. Nghĩa là, G^2 chứa một cạnh giữa u và w mỗi khi G chứa một lộ trình có chính xác hai cạnh giữa u và w. Mô tả các thuật toán hiệu quả để tính G^2 từ G với cả phép biểu diễn danh sách kề lẫn ma trận kề của G. Phân tích thời gian thực hiện của các thuật toán.

23.1-6

Khi một phép biểu diễn ma trận kề được dùng, hầu hết các thuật

toán đồ thị yêu cầu thời gian $\Theta(V^2)$, nhưng có vài ngoại lệ. Chứng tỏ tiến trình xác định một đồ thị có hướng có chứa một ***bồn*** [sink]—một đỉnh có độ-vào $|V| - 1$ và độ-ra 0—hay không có thể được xác định trong thời gian $O(V)$, cho dù dùng phép biểu diễn ma trận kề.

23.1-7

Ma trận liên thuộc [incidence matrix] của một đồ thị có hướng $G = (V, E)$ là một ma trận $|V| \times |E|$ $B = (b_{ij})$ sao cho

$$b_{ij} = \begin{cases} -1 & \text{nếu cạnh } j \text{ rời đỉnh } i, \\ 1 & \text{nếu cạnh } j \text{ nhập đỉnh } i, \\ 0 & \text{bằng không}. \end{cases}$$

Mô tả nội dung mà các khoản nhập của tích ma trận BB^T biểu diễn, ở đó B^T là chuyển vị của B.

23.2 Tìm kiếm độ rộng đầu tiên

Tìm kiếm độ rộng đầu tiên [breadth-first search] là một trong các thuật toán đơn giản nhất để tìm kiếm trong một đồ thị và là nguyên mẫu cho nhiều thuật toán đồ thị quan trọng. Thuật toán các lộ trình ngắn nhất có nguồn đơn của Dijkstra (Chương 25) và thuật toán cây tỏa nhánh cực tiểu của Prim (Đoạn 24.2) dùng các ý tưởng tương tự như trong kiểu tìm kiếm độ rộng đầu tiên.

Cho một đồ thị $G = (V, E)$ và một đỉnh ***nguồn*** được đánh dấu s, thuật toán tìm kiếm độ rộng đầu tiên khảo sát đối xứng các cạnh của G để "khám phá" mọi đỉnh khả dụng từ s. Nó tính toán khoảng cách (số cạnh ít nhất) từ s đến tất cả các đỉnh khả dụng như vậy. Nó cũng tạo ra một "cây độ rộng đầu tiên" có gốc s chứa tất cả các đỉnh khả dụng như vậy. Với một đỉnh v khả dụng từ s, lộ trình trong cây độ rộng đầu tiên từ s đến v tương ứng với một "lộ trình ngắn nhất" từ s đến v trong G, nghĩa là, một lộ trình chứa ít cạnh nhất. Thuật toán làm việc trên cả đồ thị có hướng lẫn không có hướng.

Sở dĩ gọi là tìm kiếm độ rộng đầu tiên bởi vì nó mở rộng biên giới giữa các đỉnh đã khám phá lẫn chưa khám phá một cách đồng đều qua độ rộng của biên giới. Nghĩa là, thuật toán khám phá tất cả các đỉnh tại khoảng cách k từ s trước khi khám phá bất kỳ đỉnh nào tại khoảng cách $k + 1$.

Để theo dõi tiến độ, thuật toán tìm kiếm độ rộng đầu tiên sẽ tô màu mỗi đỉnh là trắng, xám, hoặc đen. Tất cả các đỉnh bắt đầu là trắng và về sau có thể trở thành xám rồi đen. Một đỉnh ***được khám phá*** khi lần đầu

tiên gặp nó trong đợt tìm kiếm, vào lúc đó nó trở thành không trắng. Do đó, các đỉnh xám và đen đã được khám phá, nhưng thuật toán tìm kiếm độ rộng đầu tiên sẽ phân biệt giữa chúng để bảo đảm đợt tìm kiếm tiếp tục theo kiểu độ rộng đầu tiên. Nếu $(u, v) \in E$ và đỉnh u là đen, thì đỉnh v là xám hoặc đen; nghĩa là, tất cả các đỉnh kề với các đỉnh đen đều đã được khám phá. Các đỉnh xám có thể có vài các đỉnh trắng kề; chúng biểu diễn cho biên giới giữa các đỉnh đã khám phá và chưa khám phá.

Thuật toán tìm kiếm độ rộng đầu tiên kiến tạo một cây độ rộng đầu tiên, thoạt đầu chỉ chứa gốc của nó, là đỉnh nguồn s. Mỗi khi một đỉnh trắng v được khám phá trong quá trình quét danh sách kề của một đỉnh u đã khám phá, đỉnh v và cạnh (u, v) được bổ sung vào cây. Ta nói rằng u là ***phần tử tiền vị*** hoặc ***cha*** của v trong cây độ rộng đầu tiên. Bởi một đỉnh được khám phá tối đa một lần, nên nó có tối đa một cha. Các mối quan hệ tiền bối và hậu duệ trong cây độ rộng đầu tiên được định nghĩa tương đối với gốc s như thường lệ: nếu u nằm trên một lộ trình trong cây từ gốc s đến đỉnh v, thì u là một tiền bối của v và v là một hậu duệ của u.

Thủ tục tìm kiếm độ rộng đầu tiên BFS dưới đây mặc nhận đồ thị nhập liệu $G = (V, E)$ được biểu thị bằng các danh sách kề. Nó duy trì một vài cấu trúc dữ liệu bổ sung với mỗi đỉnh trong đồ thị. Màu của mỗi đỉnh $u \in V$ được lưu trữ trong biến *color*[u], và phần tử tiền vị của u được lưu trữ trong biến $\pi[u]$. Nếu u không có phần tử tiền vị (ví dụ, nếu $u = s$ hoặc u chưa được khám phá), thì $\pi[u]$ = NIL. Khoảng cách từ nguồn s đến đỉnh u mà thuật toán tính toán sẽ được lưu trữ trong $d[u]$. Thuật toán cũng sử dụng một hàng đợi vào trước ra trước Q (xem Đoạn 11.1) để quản lý tập hợp các đỉnh xám.

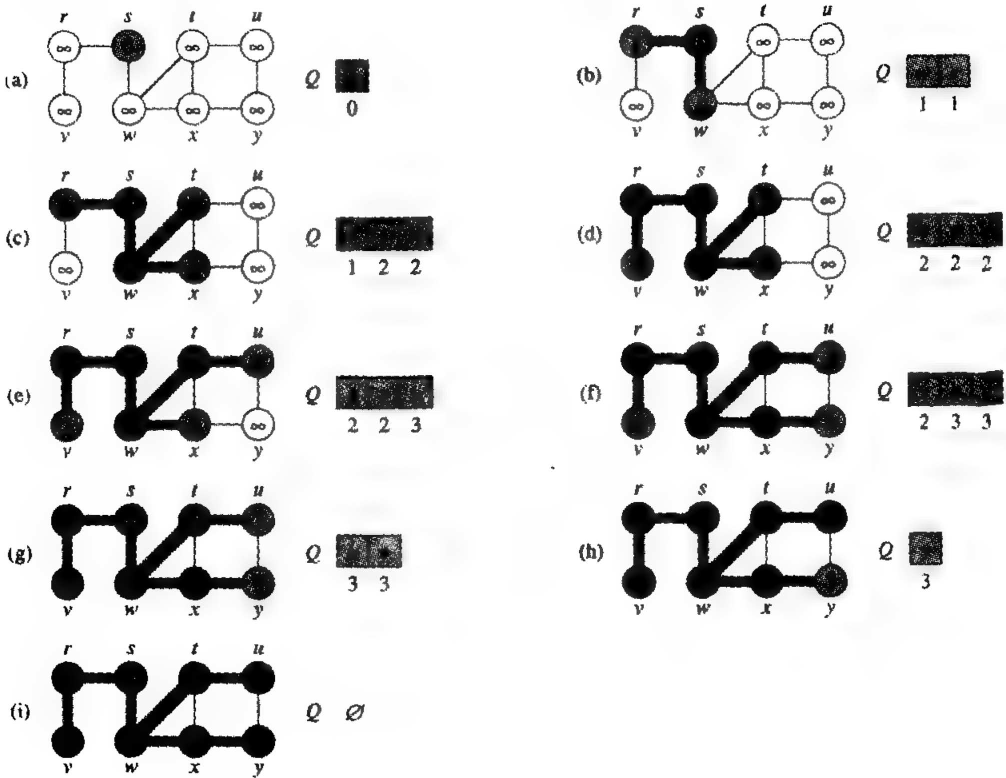
```
BFS(G, s)
for mỗi đỉnh u ∈ V[G] - {s}
     do color[u] ∈ WHITE
        d[u] ← ∞
        π[u] ← NIL
color[s] ← GRAY
d[s] ← 0
π[s] ← NIL
Q ← {s}
```

```
9   while Q ≠ ∅
10       do u ← head[Q]
11          for mỗi v ∈ Adj[u]
12              do if color[v] = WHITE
13                    then color[v] ← GRAY
14                         d[v] ← d[u] + 1
15                         π[v] ∈ u
16                         ENQUEUE(Q,v)
17          DEQUEUE(Q)
18          color[u] ← BLACK
```

Hình 23.3 minh họa tiến độ của BFS trên một đồ thị mẫu.

Thủ tục BFS làm việc như sau. Các dòng 1-4 sơn trắng mọi đỉnh, ấn định *d* [*u*] là vô cực cho mọi đỉnh *u*, và ấn định cha của mọi đỉnh là NIL. Dòng 5 sơn xám đỉnh nguồn *s*, bởi nó được xem là đã khám phá khi thủ tục bắt đầu. Dòng 6 khởi tạo *d*[*s*] theo 0, và dòng 7 ấn định phần tử tiền vị của nguồn là NIL. Dòng 8 khởi tạo *Q* theo hàng đợi chỉ chứa đỉnh *s*; sau đó, *Q* luôn chứa tập hợp các đỉnh xám.

Vòng lặp chính của chương trình được chứa trong các dòng 9-18. Vòng lặp lặp lại miễn là ở đó vẫn còn các đỉnh xám, là các đỉnh đã khám phá song các danh sách kề của chúng chưa được xem xét đầy đủ. Dòng 10 xác định đỉnh xám *u* tại đầu hàng đợi *Q*. Vòng lặp **for** của các dòng 11-16 xét mỗi đỉnh *v* trong danh sách kề của *u*. Nếu *v* là trắng, thì nó chưa được khám phá, và thuật toán khám phá nó bằng cách thi hành các dòng 13-16. Trước tiên nó được tô xám, và khoảng cách *d*[*v*] của nó được ấn định là *d* [*u*] + 1. Sau đó, *u* được ghi nhận là cha của nó. Cuối cùng, nó được đặt tại đuôi hàng đợi *Q*. Khi tất cả các đỉnh trên danh sách kề của *u* đã được xem xét, *u* được gỡ bỏ ra khỏi *Q* và được tô đen trong các dòng 17-18.



Hình 23.3 Phép toán của BFS trên một đồ thị không có hướng. Các cạnh cây được nêu ở dạng tô bóng vì BFS đã tạo chúng. Trong mỗi đỉnh u được nêu $d[u]$. Hàng đợi Q được nêu tại đầu mỗi lần lặp lại của vòng lặp **while** trong các dòng 9-18. Các khoảng cách đỉnh được nêu cạnh các đỉnh trong hàng đợi.

Phân tích

Trước khi chứng minh tất cả các tính chất khác nhau của thuật toán tìm kiếm độ rộng đầu tiên, ta tiến hành một công việc hơi dễ hơn đó là phân tích thời gian thực hiện của nó trên một đồ thị nhập liệu $G = (V, E)$. Sau khi khởi tạo, không có đỉnh nào được tô trắng, và như vậy đợt kiểm tra trong dòng 12 bảo đảm mỗi đỉnh được đưa vào hàng đợi tối đa một lần, và do đó ra khỏi hàng đợi tối đa một lần. Các phép toán đưa vào và lấy ra hàng đợi sẽ mất $O(1)$ thời gian, do đó tổng thời gian dành cho các phép toán hàng đợi là $O(V)$.

Bởi danh sách kề của mỗi đỉnh chỉ được quét khi đỉnh ra khỏi hàng đợi, nên danh sách kề của mỗi đỉnh được quét tối đa một lần. Bởi tổng các chiều dài của tất cả các danh sách kề là $\Theta(E)$, nên tối đa $O(E)$ thời gian được bỏ ra trong toàn bộ tiến trình quét các danh sách kề. Phần việc chung để khởi tạo là $O(V)$, và như vậy tổng thời gian thực hiện của BFS là $O(V + E)$. Như vậy, thuật toán tìm kiếm độ rộng đầu tiên chạy

trong thời gian tuyến tính theo kích cỡ của phép biểu diễn danh sách kề của G.

Các lộ trình ngắn nhất

Phần đầu đoạn này, ta đã xác nhận thuật toán tìm kiếm độ rộng đầu tiên tìm khoảng cách đến từng đỉnh khả dụng trong một đồ thị $G = (V, E)$ từ một đỉnh nguồn đã cho $s \in V$. Hãy định nghĩa ***khoảng cách lộ trình ngắn nhất*** $\delta(s, v)$ từ s đến v như là số cạnh tối thiểu trong bất kỳ lộ trình nào từ đỉnh s đến đỉnh v, hoặc nếu không ∞ nếu không có lộ trình nào từ s đến v. Một lộ trình có chiều dài $\delta(s, v)$ từ s đến v được xem là một ***lộ trình ngắn nhất***[1] từ s đến v. Trước khi chứng tỏ thuật toán tìm kiếm độ rộng đầu tiên thực tế tính toán các khoảng cách lộ trình ngắn nhất, ta nghiên cứu một tính chất quan trọng của các khoảng cách lộ trình ngắn nhất.

Bổ đề 23.1

Cho $G = (V, E)$ là một đồ thị có hướng hoặc không có hướng, và cho $s \in V$ là một đỉnh tùy ý. Thì, với bất kỳ cạnh $(u, v) \in E$,

$\delta(s, v) \le \delta(s, u) + 1$.

Chứng minh Nếu u khả dụng từ s, thì v cũng vậy. Trong trường hợp này, lộ trình ngắn nhất từ s đến v không thể dài hơn lộ trình ngắn nhất từ s đến u theo sau là cạnh (u, v), và như vậy bất đẳng thức đứng vững. Nếu u không khả dụng từ s, thì $\delta(s, u) = \infty$, và bất đẳng thức đứng vững.

Ta muốn chứng tỏ BFS tính toán đúng đắn $d[v] = \delta(s, v)$ với mỗi đỉnh $v \in V$. Trước tiên, ta chứng tỏ $d[v]$ định cận $\delta(s,v)$ từ bên trên.

Bổ đề 23.2

Cho $G = (V, E)$ là một đồ thị có hướng hoặc không có hướng, và giả sử BFS chạy trên G từ một đỉnh nguồn đã cho $s \in V$. Thì khi kết thúc, với mỗi đỉnh $v \in V$, giá trị $d[v]$ đã được BFS tính toán sẽ thỏa $d[v] \ge \delta(s, v)$.

Chứng minh Ta dùng phương pháp quy nạp trên số lần mà một đỉnh được đặt trong hàng đợi Q. Giả thuyết quy nạp của chúng ta đó là $d[v] \ge \delta(s, v)$ với tất cả $v \in V$.

Cơ sở của phương pháp quy nạp là tình huống ngay sau khi s được đặt trong Q trong dòng 8 của BFS. Ở đây giả thuyết quy nạp đứng vững, bởi $d[s] = 0 = \delta(s, s)$ và $d[v] = \infty \ge \delta(s, v)$ với tất cả $v \in V - \{s\}$.

---

1 Trong các Chương 25 và 26, ta sẽ tổng quát hóa cuộc nghiên cứu của chúng ta về các lộ trình ngắn nhất cho các đồ thị gia trọng, ở đó mọi cạnh có một trọng số có giá trị thực và trọng số của một lộ trình là tổng của các trọng số của các cạnh cấu thành của nó. Đồ thị được xét trong chương hiện tại là không gia trọng.

Với bước quy nạp, ta xét một đỉnh trắng v được khám phá trong đợt tìm kiếm từ một đỉnh u. Giả thuyết quy nạp hàm ý rằng $d[u] \geq \delta(s, u)$. Từ phép gán được thực hiện bởi dòng 14 và từ Bổ đề 23.1, ta được

$$\begin{aligned} d[v] &= d[u] + 1 \\ &\geq \delta(s, u) + 1 \\ &\geq \delta(s, v)\,. \end{aligned}$$

Sau đó, đỉnh v được chèn vào hàng đợi Q, và nó không bao giờ được chèn trở lại bởi nó cũng được tô xám và mệnh đề **then** của các dòng 13-16 chỉ được thi hành cho các đỉnh trắng. Như vậy, giá trị của $d[v]$ không bao giờ thay đổi lại, và giả thuyết quy nạp được duy trì.

Để chứng minh $d[v] = \delta(s, v)$, trước tiên ta phải nêu cách hàng đợi Q hoạt động trong tiến trình của BFS một cách chính xác hơn. Bổ đề kế tiếp chứng tỏ mọi lúc, ta có tối đa hai giá trị d riêng biệt trong hàng đợi.

Bổ đề 23.3

Giả sử trong khi thi hành BFS trên một đồ thị $G = (V, E)$, hàng đợi Q chứa các đỉnh $\langle v_1, v_2, \ldots, v_r \rangle$, ở đó v_1 là đầu của Q và v_r là đuôi. Như vậy, $d[v_r] \leq d[v_1] + 1$ và $d[v_i] \leq d[v_{i+1}]$ với $i = 1, 2, \ldots, r-1$.

Chứng minh Ta chứng minh bằng phương pháp quy nạp trên số lượng các phép toán hàng đợi. Thoạt đầu, khi hàng đợi chỉ chứa s, bổ đề chắc chắn đứng vững.

Với bước quy nạp, ta phải chứng minh bổ đề đứng vững sau khi một đỉnh được lấy ra khỏi hàng đợi lẫn đưa vào hàng đợi. Nếu đầu v_1 của hàng đợi ra khỏi hàng đợi, đầu mới là v_2. (Nếu hàng đợi trở nên trống, thì bổ đề đứng vững một cách trống không.) Nhưng sau đó ta có $d[v_r] \leq d[v_1] + 1 \leq d[v_2] + 1$, và các bất đẳng thức còn lại không bị tác động. Như vậy, bổ đề xảy ra với v_2 làm đầu. Tiến trình đưa một đỉnh vào hàng đợi yêu cầu xem xét mã kỹ hơn. Trong dòng 16 của BFS, khi đỉnh v được đưa vào hàng đợi, như vậy trở thành v_{r+1}, đầu v_1 của Q thực tế là đỉnh u mà danh sách kề của nó hiện đang được quét. Như vậy, $d[v_{r+1}] = d[v] = d[u] + 1 = d[v_1] + 1$. Ta cũng có $d[v_r] \leq d[v_1] + 1 = d[u] + 1 = d[v] = d[v_{r+1}]$, và các bất đẳng thức còn lại không bị tác động. Như vậy, bổ đề xảy ra khi v được đưa vào hàng đợi.

Giờ đây, ta có thể chứng minh thuật toán tìm kiếm độ rộng đầu tiên tìm các khoảng cách lộ trình ngắn nhất một cách đúng đắn.

Định lý 23.4 (Tính đúng đắn của thuật toán tìm kiếm độ rộng đầu tiên)

Cho $G = (V, E)$ là một đồ thị có hướng hoặc không có hướng, và giả

sử BFS chạy trên G từ một đỉnh nguồn đã cho $s \in V$. Như vậy, trong khi thi hành, BFS khám phá mọi đỉnh $v \in V$ là khả dụng từ nguồn s, và khi kết thúc, $d[v] = \delta(s,v)$ với tất cả $v \in V$. Hơn nữa, với bất kỳ đỉnh $v \neq s$ là khả dụng từ s, một trong các lộ trình ngắn nhất từ s đến v là lộ trình ngắn nhất từ s đến $\pi[v]$ theo sau là cạnh $(\pi[v], v)$.

Chứng minh Ta bắt đầu với trường hợp ở đó v không thể dụng từ s. Bởi Bổ đề 23.2 cho $d[v] \geq \delta(s,v) = \infty$, nên đỉnh v không thể có $d[v]$ được ấn định theo một giá trị hữu hạn trong dòng 14. Với phương pháp quy nạp, không thể có một đỉnh đầu tiên có giá trị d được dòng 14 ấn định theo ∞. Do đó, dòng 14 chỉ được thi hành cho các đỉnh có các giá trị d hữu hạn. Như vậy, nếu v không thể dụng, nó không bao giờ được khám phá.

Phần chính của chứng minh dành cho các đỉnh khả dụng từ s. Cho V_k thể hiện tập hợp các đỉnh tại khoảng cách k từ s; nghĩa là, $V_k = \{v \in V : \delta(s, v) = k\}$. Phép chứng minh tiến hành bằng phương pháp quy nạp trên k. Với tư cách là một giả thuyết quy nạp, ta mặc nhận rằng với mỗi đỉnh $v \in V_k$, ta có chính xác một điểm trong khi thi hành BFS ở đó

- v được tô xám,
- $d[v]$ được ấn định theo k,
- nếu $v \neq s$, thì $\pi[v]$ được ấn định theo u với một $u \in V_{k-1}$, và
- v được chèn vào hàng đợi Q.

Như đã nêu trên đây, chắc chắn ta có tối đa một điểm như vậy.

Cơ bản là $k = 0$. Ta có $V_0 = \{s\}$, bởi nguồn s là đỉnh duy nhất tại khoảng cách 0 từ s. Trong khi khởi tạo, s được tô xám, $d[s]$ được ấn định là 0, và s được đưa vào Q, do đó giả thuyết quy nạp đứng vững.

Với bước quy nạp, ta bắt đầu bằng cách lưu ý rằng hàng đợi Q không bao giờ trống cho đến khi thuật toán kết thúc và, một khi một đỉnh u được chèn vào hàng đợi, cả $d[u]$ lẫn $\pi[u]$ đều không hề thay đổi. Do đó, qua Bổ đề 23.3, nếu các đỉnh được chèn vào hàng đợi trong khi thuật toán thi hành theo thứ tự $v_1, v_2, \ldots, v_r$, thì dãy các khoảng cách sẽ tăng đơn điệu: $d[v_i] \leq d[v_{i+1}]$ với $i = 1, 2, \ldots, r-1$.

Giờ đây ta xét một đỉnh tùy ý $v \in V_k$, ở đó $k \geq 1$. Tính chất đơn điệu, phối hợp với $d[v] \geq k$ (theo Bổ đề 23.2) và giả thuyết quy nạp, hàm ý rằng v phải được khám phá sau khi tất cả các đỉnh trong V_{k-1} được đưa vào hàng đợi, bằng không nó không được khám phá gì cả.

Do $\delta(s, v) = k$, có một lộ trình k cạnh từ s đến v, và như vậy ở đó tồn tại một đỉnh $u \in V_{k-1}$ sao cho $(u, v) \in E$. Không làm mất đi tính tổng quát, cho u là một đỉnh đầu tiên như vậy được tô xám, nó phải xảy ra bởi,

theo phương pháp quy nạp, tất cả các đỉnh trong V_{k-1}, được tô xám. Mã của BFS lập hàng đợi mọi đỉnh được tô xám, và do đó chung cuộc u phải xuất hiện dưới dạng đầu của hàng đợi trong dòng 10. Khi u xuất hiện dưới dạng đầu, danh sách kề của nó được quét và v được khám phá. (Đỉnh v có thể chưa được khám phá trước đó, bởi nó không kề với bất kỳ đỉnh nào trong V_j với $j < k - 1$—bằng không, v có thể không thuộc về V_k—và theo giả thiết, u là đỉnh đầu tiên được khám phá trong V_{k-1}, mà v kề với nó.) Dòng 13 tô xám v, dòng 14 thiết lập $d[v] = d[u] + 1 = k$, dòng 15 ấn định $\pi[v]$ theo u, và dòng 16 chèn v vào hàng đợi. Bởi v là một đỉnh tùy ý trong V_k, nên giả thuyết quy nạp được chứng minh.

Để kết luận chứng minh của bổ đề, ta nhận thấy nếu $v \in V_k$, thì theo như những gì ta vừa thấy, $\pi[v] \in V_{k-1}$. Như vậy, ta có thể có được một lộ trình ngắn nhất từ s đến v bằng cách lấy một lộ trình ngắn nhất từ s đến $\pi[v]$ rồi băng ngang cạnh $(\pi[v], v)$.

Các cây độ rộng đầu tiên

Thủ tục BFS xây dựng một cây độ rộng đầu tiên khi nó tìm kiếm trong đồ thị, như minh họa trong Hình 23.3. Cây được biểu thị bởi trường π trong mỗi đỉnh. Chính thức hơn, với một đồ thị $G = (V, E)$ có nguồn s, ta định nghĩa ***đồ thị con phần tử tiền vị*** của G là $G_\pi = (V_\pi, E_\pi)$, ở đó

$$V_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\}$$

và

$$E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\} .$$

Đồ thị con phần tử tiền vị [predecessor subgraph] G_π là một ***cây độ rộng đầu tiên*** nếu V_π bao gồm các đỉnh khả dụng từ s và, với tất cả $v \in V_\pi$, ta có một lộ trình đơn giản duy nhất từ s đến v trong G_π, cũng là một lộ trình ngắn nhất từ s đến v trong G. Một cây độ rộng đầu tiên thực tế là một cây, bởi nó liên thông và $|E_\pi| = |V_\pi| - 1$ (xem Định lý 5.2). Các cạnh trong E_π được gọi là ***các cạnh cây***.

Sau khi chạy BFS từ một nguồn s trên một đồ thị G, bổ đề dưới đây chứng tỏ đồ thị con phần tử tiền vị là một cây độ rộng đầu tiên.

Bổ đề 23.5

Khi được áp dụng cho một đồ thị có hướng hoặc không có hướng $G = (V, E)$, thủ tục BFS kiến tạo π sao cho đồ thị con phần tử tiền vị $G_\pi = (V_\pi, E_\pi)$ là một cây độ rộng đầu tiên.

Chứng minh Dòng 15 của BFS chỉ ấn định $\pi[v] = u$ nếu $(u, v) \in E$ và $\delta(s, v) \leq \infty$ —nghĩa là, nếu v khả dụng từ s—và như vậy V_π bao gồm các

đỉnh trong V khả dụng từ v. Do G_π hình thành một cây, nên nó chứa một lộ trình duy nhất từ s đến mỗi đỉnh trong V_π. Nhờ áp dụng Định lý 23.4 theo quy nạp, ta kết luận mọi lộ trình như vậy là một lộ trình ngắn nhất.

Thủ tục dưới đây in ra các đỉnh trên một lộ trình ngắn nhất từ s đến v, giả định BFS đã chạy để tính toán cây lộ trình ngắn nhất.

```
PRINT-PATH(G, s, v)
if v = s
   then in s
   else if π[v] = NIL
           then print "không có lộ trình từ" s "đến" v "tồn tại"
           else PRINT-PATH(G, s, π[v])
                print v
```

Thủ tục này chạy trong thời gian tuyến tính theo số lượng các đỉnh trong lộ trình đã in, bởi mỗi lệnh gọi đệ quy dành cho một lộ trình ngắn hơn một đỉnh.

Bài tập

23.2-1

Nêu kết quả của việc chạy thuật toán tìm kiếm độ rộng đầu tiên trên đồ thị có hướng của Hình 23.2(a), dùng đỉnh 3 làm nguồn.

23.2-2

Nêu kết quả của việc chạy thuật toán tìm kiếm độ rộng đầu tiên trên đồ thị không có hướng của Hình 23.3, dùng đỉnh u làm nguồn.

23.2-3

Nêu thời gian thực hiện của BFS nếu đồ thị nhập liệu củanó được biểu thị bởi một ma trận kề và thuật toán được sửa đổi để điều quản dạng nhập liệu này?

23.2-4

Chứng tỏ trong một thuật toán tìm kiếm độ rộng đầu tiên, giá trị d[u] được gán cho một đỉnh u độc lập với thứ tự ở đó các đỉnh trong mỗi danh sách kề được cho.

23.2-5

Nêu một ví dụ của một đồ thị có hướng $G = (V, E)$, một đỉnh nguồn $s \in V$, và một tập hợp các cạnh cây $E_\pi \subseteq E$ sao cho với mỗi đỉnh $v \in V$, lộ trình duy nhất trong E_π từ s đến v là một lộ trình ngắn nhất trong G,

hơn nữa không thể tạo tập hợp các cạnh E_π bằng cách chạy BFS trên G, bất kể cách sắp xếp các đỉnh trong mỗi danh sách kề.

23.2-6

Nêu một thuật toán hiệu quả để xác định xem một đồ thị không có hướng có phải là hai nhánh không.

23.2-7 *

Đường kính của một cây $T = (V, E)$ được cho bởi

$$\max_{u,v \in V} \delta(u, v)\ ;$$

nghĩa là, đường kính là lớn nhất trong tất cả các khoảng cách lộ trình ngắn nhất trong cây. Nêu một thuật toán hiệu quả để tính toán đường kính của một cây, và phân tích thời gian thực hiện của thuật toán.

23.2-8

Cho $G = (V, E)$ là một đồ thị không có hướng. Nêu một thuật toán $O(V + E)$-thời gian để tính toán một lộ trình trong G băng ngang mỗi cạnh trong E chính xác một lần theo mỗi hướng. Mô tả cách tìm cách thoát ra khỏi một mê cung nếu bạn có một nguồn cung cấp penni lớn.

23.3 Tìm kiếm độ sâu đầu tiên

Chiến lược mà thuật toán tìm kiếm độ sâu đầu tiên tuân thủ đó là, giống như tên gọi của nó hàm ý, để tìm kiếm "sâu hơn" trong đồ thị nếu có thể. Trong thuật toán tìm kiếm độ sâu đầu tiên, các cạnh được khảo sát từ đỉnh mới được khám phá v vẫn có các cạnh chưa khảo sát rời nó. Sau khi khảo sát tất cả các cạnh của v, đợt tìm kiếm "rà ngược" để khảo sát các cạnh rời đỉnh mà từ đó v đã được khám phá. Tiến trình này tiếp tục cho đến khi ta có tất cả các đỉnh đã khám phá khả dụng từ đỉnh nguồn ban đầu. Nếu vẫn còn các đỉnh chưa khám phá, thì một trong số chúng được chọn làm một nguồn mới và đợt tìm kiếm được lặp lại từ nguồn đó. Nguyên cả tiến trình này được lặp lại cho đến khi tất cả các đỉnh đều được khám phá.

Giống như trong tìm kiếm độ rộng đầu tiên, mỗi khi một đỉnh v được khám phá trong một quét danh sách kề của một đỉnh u đã khám phá, thuật toán tìm kiếm độ sâu đầu tiên ghi nhận sự kiện này bằng cách ấn định trường tiền vị của v $\pi[v]$ thành u. Khác với thuật toán tìm kiếm độ rộng đầu tiên, ở đó đồ thị con phần tử tiền vị của nó hình thành một cây, đồ thị con phần tử tiền vị mà một phép tìm kiếm độ sâu đầu tiên tạo ra có thể bao gồm vài cây, bởi đợt tìm kiếm có thể được lặp lại từ nhiều nguồn. Do đó, ***đồ thị con phần tử tiền vị*** [predecessor subgraph]

của một tìm kiếm độ sâu đầu tiên được định nghĩa hơi khác với trường hợp của tìm kiếm độ rộng đầu tiên: ta cho $G_\pi = (V, E_\pi)$, ở đó

$$E_\pi = \{(\pi[v], v) : v \in V \text{ và } \pi[v] \neq \text{NIL}\} .$$

Đồ thị con phần tử tiền vị của một đợt tìm kiếm độ sâu đầu tiên hình thành một ***rừng độ sâu đầu tiên*** bao gồm một số ***cây độ sâu đầu tiên.*** Các cạnh trong E_π được gọi là ***các cạnh cây.***

Giống như trong thuật toán tìm kiếm độ rộng đầu tiên, các đỉnh được tô màu trong đợt tìm kiếm để nêu rõ trạng thái của chúng. Thoạt đầu mỗi đỉnh là trắng, được tô xám khi nó ***được khám phá*** trong đợt tìm kiếm, và được tô đen khi nó ***hoàn tất,*** nghĩa là, khi danh sách kề củanó đã được xem xét hoàn toàn. Kỹ thuật này bảo đảm mỗi đỉnh kết thúc chính xác bằng một cây độ sâu đầu tiên, sao cho các cây này rời nhau.

Ngoài việc tạo một rừng độ sâu đầu tiên, thuật toán tìm kiếm độ sâu đầu tiên cũng ***gán tem thời gian*** cho mỗi đỉnh. Mỗi đỉnh v có hai tem thời gian: tem thời gian đầu tiên $d[v]$ ghi nhận thời gian v được khám phá lần đầu tiên (và được tô xám), và tem thời gian thứ hai $f[v]$ ghi nhận thời gian đợt tìm kiếm hoàn tất việc xét danh sách kề của v (và tô đen v). Các tem thời gian này được dùng trong nhiều thuật toán đồ thị và nói chung là hữu ích cho việc biện luận về cách ứng xử của thuật toán tìm kiếm độ sâu đầu tiên.

Thủ tục DFS dưới đây ghi nhận thời gian nó khám phá đỉnh u trong biến $d[u]$ và thời gian nó hoàn tất đỉnh u trong biến $f[u]$. Các tem thời gian này là các số nguyên giữa 1 và $2|V|$, bởi có một sự kiện khám phá và một sự kiện hoàn tất cho từng đỉnh $|V|$. Với mọi đỉnh u,

$$d[u] < f[u] . \qquad (23.1)$$

Đỉnh u là WHITE trước thời gian $d[u]$, GRAY giữa thời gian $d[u]$ và thời gian $f[u]$, và sau đó là BLACK.

Mã giả dưới đây là thuật toán tìm kiếm độ sâu đầu tiên cơ bản. Đồ thị nhập liệu G có thể không có hướng hoặc có hướng. Biến *time* là một biến toàn cục mà ta dùng để gán tem thời gian.

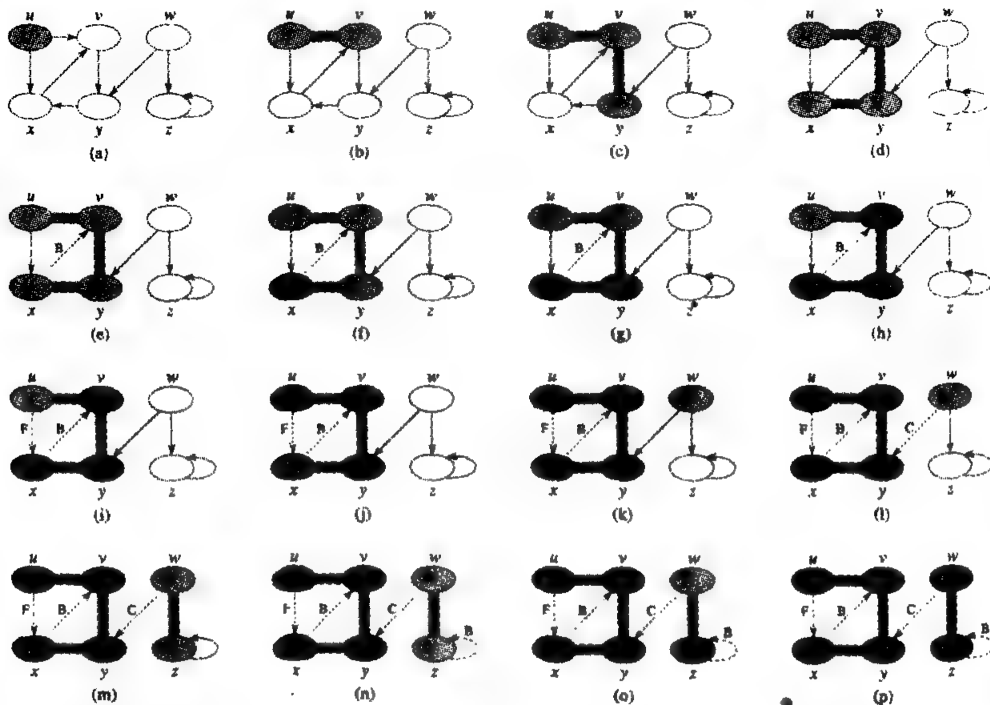
```
DFS(G)
for mỗi đỉnh u ∈ V[G]
      do color[u] ← WHITE
         π[u] ← NIL
time ← 0
for mỗi đỉnh u ∈ V[G]
      do if color[u] = WHITE
            then DFS-VISIT(u)
```

DFS-VISIT(u)

```
1  color[u] ← GRAY                ▷ Đỉnh trắng u vừa được khám phá.
2  d [u] ← time ← time + 1
3  for mỗi v ∈ Adj[u]             ▷ Khảo sát cạnh (u, v).
4       do if color[v] = WHITE
5            then π[v] ← u
6                 DFS-VISIT(v)
7  color[u] - BLACK               ▷ Tô đen u; nếu nó hoàn tất.
8  f[u] ← time ← time + 1
```

Hình 23.4 minh họa tiến độ của DFS trên đồ thị nêu trong Hình 23.2.

Thủ tục DFS làm việc như sau. Các dòng 1-3 sơn trắng tất cả các đỉnh và khởi tạo các trường π của chúng theo NIL. Dòng 4 chỉnh lại bộ đếm thời gian toàn cục. Các dòng 5-7 lần lượt kiểm tra mỗi đỉnh trong V và, khi tìm thấy một đỉnh trắng, sẽ ghé thăm nó bằng DFS-VISIT. Mỗi lần DFS-VISIT(u) được gọi trong dòng 7, đỉnh u trở thành gốc của một cây mới trong rừng độ sâu đầu tiên. Khi DFS trả về, mọi đỉnh u đã được gán một thời gian khám phá $d[u]$ và một thời gian kết thúc $f[u]$.



Hình 23-4 Tiến độ của thuật toán tìm kiếm độ sâu đầu tiên DFS trên một đồ thị có hướng. Khi thuật toán khảo sát các cạnh, chúng được nêu dưới dạng tô bóng (nếu là các cạnh cây) hoặc chấm cách (các trường hợp khác). Các cạnh không thuộc cây được gán nhãn B, C, hoặc F tùy theo chúng là các cạnh phía sau, chéo, phía trước. Các đỉnh được gán tem thời gian bằng thời gian khám phá/thời gian hoàn tất.

Trong mỗi lần gọi DFS-VISIT(u), thoạt đầu đỉnh u là trắng. Dòng 1 sơn xám u, và dòng 2 ghi nhận thời gian khám phá $d[u]$ bằng cách gia số và lưu biến toàn cục *time*. Các dòng 3-6 xét mỗi đỉnh v kề với u và ghé thăm v một cách đệ quy nếu nó là trắng. Vì mỗi đỉnh $v \in Adj[u]$ được xét trong dòng 3, ta nói rằng cạnh (u, v) được ***khảo sát*** bởi đợt tìm kiếm độ sâu đầu tiên. Cuối cùng, sau khi khảo sát mọi cạnh rời u, các dòng 7-8 sơn đen u và ghi nhận thời gian kết thúc trong $f[u]$.

Đâu là thời gian thực hiện của DFS? Các vòng lặp trên các dòng 1-2 và các dòng 5-7 của DFS mất thời gian $\Theta(V)$, không kể thời gian thi hành các lệnh gọi DFS-VISIT. Thủ tục DFS-VISIT được gọi chính xác một lần cho mỗi đỉnh $v \in V$, bởi DFS-VISIT chỉ được triệu gọi trên các đỉnh trắng và việc đầu tiên mà nó thực hiện đó là sơn xám đỉnh. Trong khi thi hành DFS-VISIT(v), vòng lặp trên các dòng 3-6 được thi hành $|Adj[v]|$ lần. Bởi

$$\sum_{v \in V} |Adj[v]| = \Theta(E),$$

nên tổng mức hao phí thi hành các dòng 2-5 của DFS-VISIT là $\Theta(E)$. Do đó, thời gian thực hiện của DFS là $\Theta(V + E)$.

Các tính chất của thuật toán tìm kiếm độ sâu đầu tiên

Tìm kiếm độ sâu đầu tiên mang lại nhiều thông tin về cấu trúc của một đồ thị. Có lẽ tính chất cơ bản nhất của thuật toán tìm kiếm độ sâu đầu tiên đó là đồ thị con phần tử tiền vị G_π sẽ hình thành một rừng các cây, bởi cấu trúc của các cây độ sâu đầu tiên chính xác soi gương cấu trúc của các lệnh gọi đệ quy của DFS-VISIT. Nghĩa là, $u = \pi[v]$ nếu và chỉ nếu DFS-VISIT(v) đã được gọi trong một đợt tìm kiếm danh sách kề của u.

Một tính chất quan trọng khác của thuật toán tìm kiếm độ sâu đầu tiên đó là các thời gian khám phá và hoàn tất có ***cấu trúc ngoặc đơn***. Nếu ta biểu diễn sự khám phá đỉnh u bằng một dấu ngoặc đơn trái "(u" và biểu diễn việc hoàn tất củanó bằng một dấu ngoặc đơn phải "u)," thì lịch sử các khám phá và hoàn tất sẽ thực hiện một biểu thức có tổ chức theo nghĩa các dấu ngoặc đơn được lồng đúng đắn. Ví dụ, việc tìm kiếm độ sâu đầu tiên của Hình 23.5(a) tương ứng với phép ngoặc đơn nêu trong Hình 23.5(b). Định lý dưới đây nêu một cách khác để phát biểu điều kiện của cấu trúc dấu ngoặc đơn.

***Định lý* 23.6 (*Định lý dấu ngoặc đơn*)**

Trong mọi tìm kiếm độ sâu đầu tiên của một đồ thị (có hướng hoặc không có hướng) $G = (V, E)$, với bất kỳ hai đỉnh u và v, chính xác sẽ có một trong ba điều kiện dưới đây:

- các quãng $[d[u], f[u]]$ và $[d[v], f[v]]$ hoàn toàn rời nhau,
- quãng $[d[u], f[u]]$ được chứa hoàn toàn trong quãng $[d[v], f[v]]$, và u là một hậu duệ của v trong cây độ sâu đầu tiên, hoặc
- quãng $[d[v], f[v]]$ được chứa hoàn toàn trong quãng $[d[u], f[u]]$, và v là một hậu duệ của u trong cây độ sâu đầu tiên.

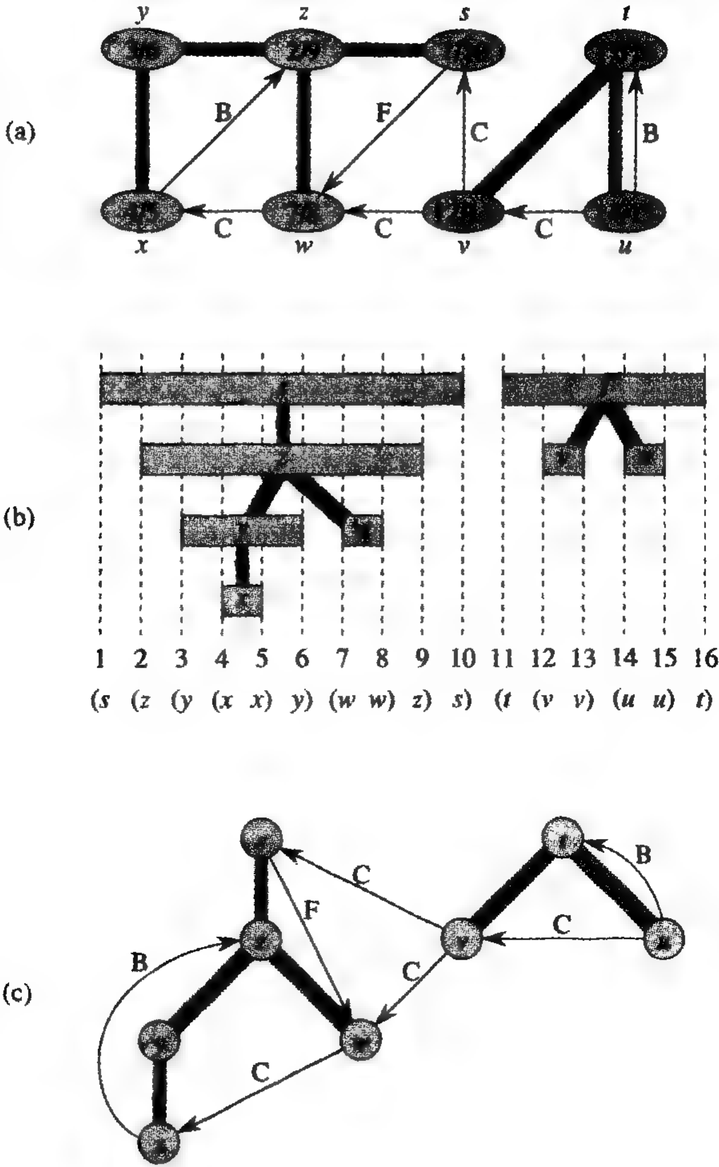
Chứng minh Ta bắt đầu với trường hợp ở đó $d[u] < d[v]$. Có hai trường hợp con để xem xét, dẫu theo $d[v] < f[u]$ hay không. Trong trường hợp con đầu tiên, $d[v] < f[u]$, do đó v đã được khám phá trong khi u vẫn còn xám. Điều này hàm ý v là một hậu duệ của u. Hơn nữa, bởi v đã được khám phá gần hơn u, nên tất cả các cạnh đi ra củanó đều được khảo sát, và v được hoàn tất, trước khi đợt tìm kiếm trả về và hoàn tất u. Do đó, trong trường hợp này, quãng $[d[v], f[v]]$ hoàn toàn nằm trong quãng $[d[u], f[u]]$. Trong trường hợp con thứ hai, $f[u] < d[v]$, và bất đẳng thức (23.1) hàm ý các quãng $[d[u],f[u]]$ và $[d[v], f[v]]$ rời nhau.

Trường hợp ở đó $d[v] < d[u]$ cũng tương tự, với các vai trò của u và v được đảo ngược trong đối số trên đây.

***Hệ luận* 23.7 (*Lồng ghép các quãng của các hậu duệ*)**

Đỉnh v là một hậu duệ riêng của đỉnh u trong rừng độ sâu đầu tiên với một đồ thị G (có hướng hoặc không có hướng) nếu và chỉ nếu $d[u] < d[v] < f[v] < f[u]$.

Chứng minh Tức thời từ Định lý 23.6.



Hình 23.5 Các tính chất của thuật toán tìm kiếm độ sâu đầu tiên. **(a)** Kết quả của một đợt tìm kiếm độ sâu đầu tiên của một đồ thị có hướng. Các đỉnh được gán tem thời gian và các kiểu cạnh được nêu như trong Hình 23.4. **(b)** Các quãng cho thời gian khám phá và thời gian kết thúc của mỗi đỉnh tương ứng với phép ngoặc đơn đã nêu. Mỗi hình chữ nhật trải dài quãng của các thời gian khám phá và hoàn tất của đỉnh tương ứng. Các cạnh cây được nêu. Nếu hai quãng phủ chồng, thì chúng sẽ lồng nhau, và đỉnh tương ứng với quãng nhỏ hơn sẽ là một hậu duệ của đỉnh tương ứng với quãng lớn hơn. **(c)** Đồ thị của phần (a) được vẽ lại với tất cả các cạnh tới và cạnh cây đổ xuống trong một cây độ sâu đầu tiên và tất cả các cạnh lùi đi lên từ một hậu duệ đến một tiền bối.

Định lý dưới đây đưa ra một đặc trưng quan trọng khác về thời gian một đỉnh là một hậu duệ của một đỉnh khác trong rừng độ sâu đầu tiên.

Định lý 23.8 (Định lý lộ trình trắng)

Trong một rừng độ sâu đầu tiên của một đồ thị $G = (V, E)$ (có hướng hoặc không có hướng), đỉnh v là một hậu duệ của đỉnh u nếu và chỉ nếu vào thời gian $d[u]$ mà đợt tìm kiếm khám phá u, đỉnh v có thể đụng từ u dọc theo một lộ trình hoàn toàn bao gồm các đỉnh trắng.

Chứng minh ⇒: Giả sử v là một hậu duệ của u. Cho w là một đỉnh bất kỳ trên lộ trình giữa u và v trong cây độ sâu đầu tiên, sao cho w là một hậu duệ của u. Theo Hệ luận 23.7, $d[u] < d[w]$, và do đó w là trắng vào thời gian $d[u]$.

⇐: Giả sử rằng đỉnh v khả đụng từ u dọc theo một lộ trình các đỉnh trắng vào thời gian $d[u]$, nhưng v không trở thành một hậu duệ của u trong cây độ sâu đầu tiên. Không để mất tính tổng quát, ta mặc nhận rằng mọi đỉnh khác dọc theo lộ trình đều trở thành một hậu duệ của u. (Bằng không, cho v là đỉnh sát nhất với u dọc theo lộ trình không trở thành một hậu duệ của u.) Cho w là phần tử tiền vị của v trong lộ trình, sao cho w là một hậu duệ của u (thực tế w và u có thể là cùng một đỉnh) và, theo Hệ luận 23.7, $f[w] \leq f[u]$. Lưu ý, v phải được khám phá sau khi u được khám phá, nhưng trước khi hoàn tất w. Do đó, $d[u] < d[v] < f[w] \leq f[u]$. Như vậy, Định lý 23.6 hàm ý rằng quãng $[d[v], f[v]]$ được chứa hoàn toàn trong quãng $[d[u], f[u]]$. Theo Hệ luận 23.7, cuối cùng v phải là một hậu duệ của u.

Phân loại các cạnh

Một tính chất thú vị khác của thuật toán tìm kiếm độ sâu đầu tiên đó là đợt tìm kiếm có thể được dùng để phân loại các cạnh của đồ thị nhập liệu $G = (V, E)$. Kiểu phân loại cạnh này có thể được dùng để lượm lặt thông tin quan trọng về một đồ thị. Ví dụ, trong đoạn kế tiếp, ta sẽ thấy một đồ thị có hướng là phi chu trình nếu và chỉ nếu một đợt tìm kiếm độ sâu đầu tiên không cho ra các cạnh "lùi" (Bổ đề 23.10).

Ta có thể định nghĩa bốn kiểu cạnh theo dạng rừng độ sâu đầu tiên G_π được tạo bởi một đợt tìm kiếm độ sâu đầu tiên trên G.

1. ***Các cạnh cây*** [tree edges] là các cạnh trong rừng độ sâu đầu tiên G_π. Cạnh (u, v) là một cạnh cây nếu v được khám phá đầu tiên bằng cách khảo sát cạnh (u, v).

2. ***Các cạnh lùi*** [back edges] là các cạnh (u, v) nối một đỉnh u với một tiền bối v trong một cây độ sâu đầu tiên. Các vòng tự lặp được xem là các cạnh lùi.

3. ***Các cạnh tới*** [forward edges] là các cạnh phi cây (u, v) nối một đỉnh u với một hậu duệ v trong một cây độ sâu đầu tiên.

4. ***Các cạnh chéo*** là tất cả các cạnh khác. Chúng có thể tiến giữa các đỉnh trong cùng cây độ sâu đầu tiên, miễn là một đỉnh không phải là tiền bối của đỉnh kia, hoặc chúng có thể tiến giữa các đỉnh trong các cây độ sâu đầu tiên khác nhau.

Trong các Hình 23.4 và 23.5, các cạnh được gán nhãn để nêu rõ kiểu của chúng. Hình 23.5(c) cũng nêu cách vẽ lại đồ thị của Hình 23.5(a) sao cho tất cả các cạnh tới và cạnh cây đổ đầu xuống trong một cây độ sâu đầu tiên và tất cả các cạnh lùi đi lên. Mọi đồ thị đều có thể vẽ lại theo cách này.

Có thể sửa đổi thuật toán DFS để phân loại các cạnh khi nó gặp chúng. Ý tưởng chính đó là có thể phân loại mỗi cạnh (u, v) theo màu của đỉnh v đã đụng khi cạnh được khảo sát lần đầu (ngoại trừ các cạnh tới và cạnh chéo không được phân biệt):

1. WHITE nêu rõ một cạnh cây,
2. GRAY nêu rõ một cạnh lùi, và
3. BLACK nêu rõ một cạnh chéo và cạnh tới.

Trường hợp thứ nhất là tức thời từ định chuẩn của thuật toán. Với trường hợp thứ hai, ta nhận thấy các đỉnh xám luôn hình thành một xích tuyến tính gồm các hậu duệ tương ứng với ngăn xếp các lần gọi DFS-VISIT hoạt động; số lượng các đỉnh xám nhiều hơn một so với độ sâu trong rừng độ sâu đầu tiên của đỉnh mới được khám phá. Việc khảo sát luôn tiến hành từ đỉnh xám sâu nhất, do đó cạnh đụng một đỉnh xám khác sẽ đụng một tiền bối. Trường hợp thứ ba điều quản khả năng còn lại; nó có thể được chứng tỏ một cạnh (u, v) như vậy là một cạnh tới nếu $d[u] < d[v]$ và là một cạnh chéo nếu $d[u] > d[v]$. (Xem Bài tập 23.3-4.)

Trong loại đồ thị không có hướng, có thể có một dạng mơ hồ nào đó trong kiểu phân loại, bởi (u, v) và (v, u) thực tế là cùng một cạnh. Trong trường hợp như vậy, cạnh được phân loại dưới dạng kiểu *đầu tiên* trong danh sách phân loại được áp dụng. Một cách tương tự (xem Bài tập 23.3-5), cạnh được phân loại theo bất kỳ hoặc (u, v) hoặc (v, u) được gặp đầu tiên trong khi thi hành thuật toán.

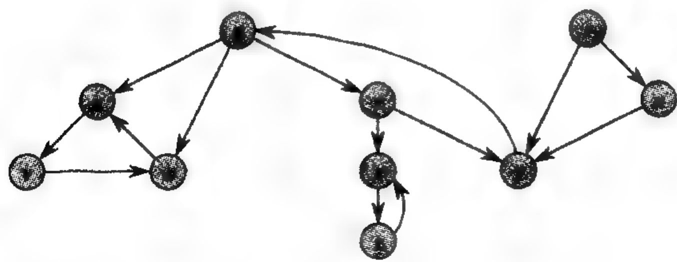
Giờ đây ta chứng tỏ các cạnh tới và cạnh chéo không bao giờ xảy ra trong một đợt tìm kiếm độ sâu đầu tiên của một đồ thị không có hướng.

Định lý 23.9

Trong một đợt tìm kiếm độ sâu đầu tiên của một đồ thị không có hướng G, mọi cạnh của G là một cạnh cây hoặc một cạnh lùi.

Chứng minh Cho (u, v) là một cạnh tùy ý của G, và không làm mất tính tổng quát, ta giả sử rằng $d[u] < d[v]$. Thì, v phải được khám phá và hoàn tất trước khi ta hoàn tất u, bởi v nằm trên danh sách kề của u. Nếu cạnh (u, v) được khảo sát đầu tiên theo hướng từ u đến v, thì (u, v) trở thành một cạnh cây. Nếu (u, v) được khảo sát đầu tiên theo hướng từ v đến u, thì (u, v) là một cạnh lùi, bởi u vẫn xám vào thời gian cạnh được khảo sát lần đầu tiên.

Ta sẽ thấy nhiều ứng dụng của các định lý này trong các đoạn sau.



Hình 23.6 Một đồ thị có hướng để dùng trong Bài tập 23.3-2 và 23.5-2.

Bài tập

23.3-1

Tạo một đồ thị 3 x 3 có các nhãn hàng và cột WHITE, GRAY, và BLACK. Trong mỗi ô (i, j), nêu rõ, tại một điểm bất kỳ trong một đợt tìm kiếm độ sâu đầu tiên của một đồ thị có hướng, có thể có một cạnh từ một đỉnh của màu i đến một đỉnh của màu j hay không. Với mỗi cạnh khả dĩ, hãy nêu rõ nó có thể có những kiểu cạnh nào. Tạo một đồ thị thứ hai như vậy cho đợt tìm kiếm độ sâu đầu tiên của một đồ thị không có hướng..

23.3-2

Nêu cách làm việc của thuật toán tìm kiếm độ sâu đầu tiên trên đồ thị của Hình 23.6. Giả sử vòng lặp **for** trong các dòng 5-7 của thủ tục DFS xem xét các đỉnh theo thứ tự abc, và mặc nhận rằng mỗi danh sách kề được sắp xếp theo thứ tự abc. Nêu các thời gian khám phá và hoàn tất cho mỗi đỉnh, và nêu cách phân loại của mỗi cạnh.

23.3-3

Nêu cấu trúc dấu ngoặc đơn của đợt tìm kiếm độ sâu đầu tiên nêu trong Hình 23.4.

23.3-4

Chứng tỏ cạnh (u, v) là

a. một cạnh cây hoặc cạnh tới nếu và chỉ nếu $d[u] < d[v] < f[v] < f[u]$,

b. một cạnh lùi nếu và chỉ nếu $d[v] < d[u] < f[u] < f[v]$, và

c. một cạnh chéo nếu và chỉ nếu $d[v] < f[v] < d[u] < f[u]$.

23.3-5

Chứng tỏ trong một đồ thị không có hướng, khi một cạnh cây (u, v) hoặc một cạnh lùi theo (u, v) hoặc (v, u) được gặp đầu tiên trong đợt tìm kiếm độ sâu đầu tiên, việc phân loại một cạnh tương đương với việc phân loại nó theo mức ưu tiên của các kiểu trong lược đồ phân loại.

23.3-6

Nêu một ví dụ ngược lại với giả định rằng nếu có một lộ trình từ u đến v trong một đồ thị có hướng G, và nếu $d[u] < d[v]$ trong một đợt tìm kiếm độ sâu đầu tiên của G, thì v là một hậu duệ của u trong rừng độ sâu đầu tiên được tạo.

23.3-7

Sửa đổi mã giả cho thuật toán tìm kiếm độ sâu đầu tiên sao cho nó in ra mọi cạnh trong đồ thị có hướng G, chung với kiểu của nó. Nêu các sửa đổi, nếu có, phải được thực hiện nếu G không có hướng.

23.3-8

Giải thích cách kết thúc khả dĩ của một đỉnh u của một đồ thị có hướng trong một cây độ sâu đầu tiên chứa chỉ u, cho dù u đều có cả các cạnh vào và ra trong G.

23.3-9

Chứng tỏ một tìm kiếm độ sâu đầu tiên của một đồ thị không có hướng G có thể được dùng để định danh các thành phần liên thông của G, và rừng độ sâu đầu tiên chứa số lượng cây ngang bằng với G có các thành phần liên thông. Nói một cách chính xác hơn, nêu cách sửa đổi thuật toán tìm kiếm độ sâu đầu tiên sao cho mỗi đỉnh v được gán một nhãn số nguyên $cc[v]$ giữa 1 và k, ở đó k là số lượng các thành phần liên thông của G, sao cho $cc[u] = cc[v]$ nếu và chỉ nếu u *và* v nằm trong cùng thành phần liên thông.

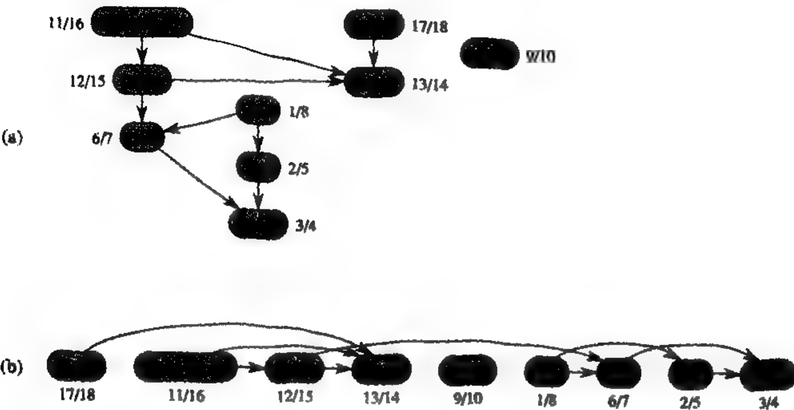
23.3-10*

Một đồ thị có hướng $G = (V, E)$ ***được nối đơn lẻ*** nếu $u \leadsto v$ hàm ý có tối đa một lộ trình đơn giản từ u đến v với tất cả các đỉnh $u, v \in V$. Nêu một thuật toán hiệu quả để xác định một đồ thị có hướng có được nối đơn lẻ hay không.

23.4 Sắp xếp theo tôpô

Đoạn này nêu cách dùng thuật toán tìm kiếm độ sâu đầu tiên để thực hiện các đợt sắp xếp tôpô của đồ thị phi chu trình có hướng [directed acyclic graphs], hoặc như trong tiếng Anh thường gọi tắt là "dag." Một ***đợt sắp xếp tôpô*** của một dag $G = (V, E)$ là một kiểu sắp xếp thứ tự tuyến tính tất cả các đỉnh củanó sao cho nếu G chứa một cạnh (u, v), thì u xuất hiện trước v trong tiến trình sắp xếp thứ tự. (Nếu đồ thị không phải phi chu trình, thì không thể có cách sắp xếp thứ tự tuyến tính.) Một đợt sắp xếp tôpô của một đồ thị có thể được xem như một tiến trình sắp xếp thứ tự các đỉnh củanó dọc theo một đường ngang sao cho tất cả các cạnh có hướng đi từ trái qua phải. Như vậy, sắp xếp tôpô khác với kiểu "sắp xếp" bình thường đã nghiên cứu trong Phần II.

Đồ thị phi chu trình có hướng được dùng trong nhiều ứng dụng để nêu rõ các mức ưu tiên giữa các sự kiện. Hình 23.7 có nêu một ví dụ nảy sinh khi Giáo sư Bumstead mặc đồ vào buổi sáng. Giáo sư phải mặc một số đồ mặc trước các đồ khác (ví dụ, mang vớ trước xỏ giày). Các mục khác có thể mặc theo thứ tự bất kỳ (ví dụ, vớ và quần dài). Một cạnh có hướng (u, v) trong dag của Hình 23.7(a) nêu rõ đồ mặc u phải được mặc trước đồ v. Do đó một đợt sắp xếp tôpô của dag này cho ta một thứ tự về mặc đồ. Hình 23.7(b) nêu dag được sắp xếp theo tôpô dưới dạng một tiến trình sắp xếp các đỉnh dọc theo một đường ngang sao cho tất cả các cạnh có hướng đi từ trái qua phải.



Hình 23.7 (a) Giáo sư Bumstead sắp xếp theo tôpô quần áo của mình khi mặc đồ. Mỗi cạnh có hướng (u, v) có nghĩa là đồ mặc u phải được mặc trước đồ v. Thời gian khám phá và hoàn tất từ một đợt tìm kiếm độ sâu đầu tiên được nêu cạnh mỗi đỉnh. **(b)** Cùng đồ thị được nêu dưới dạng đã sắp xếp theo tôpô. Các đỉnh của nó được dàn xếp từ trái qua phải theo thứ tự thời gian kết thúc giảm dần. Lưu ý, tất cả các cạnh có hướng đi từ trái qua phải.Hình 23.7(b) nêu cách thức mà các đỉnh được sắp xếp theo tôpô xuất hiện theo thứ tự đảo ngược của các thời gian hoàn tất.

Thuật toán đơn giản dưới đây sắp xếp một đồ thị phi chu trình có hướng theo tôpô.

TOPOLOGICAL-SORT (G)

1 gọi DFS(G) để tính toán các thời gian hoàn tất $f[v]$ cho mỗi đỉnh v

2 khi từng đỉnh hoàn tất, chèn nó vào trước một danh sách nối kết

3 **trả về [return]** danh sách nối kết các đỉnh

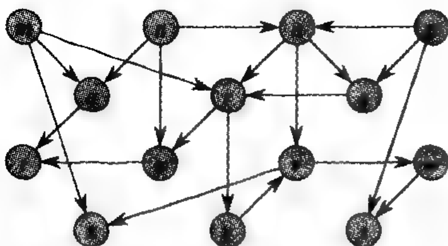
Ta có thể thực hiện một đợt sắp xếp tôpô trong thời gian $\Theta(V + E)$, bởi thuật toán tìm kiếm độ sâu đầu tiên kéo dài $\Theta(V + E)$ thời gian và mất $O(1)$ thời gian để chèn từng đỉnh $|V|$ lên đầu danh sách nối kết.

Ta chứng minh tính đúng đắn của thuật toán này bằng bổ đề chính sau đây mô tả đặc điểm của đồ thị phi chu trình có hướng.

Bổ đề 23.10

Một đồ thị có hướng G là phi chu trình nếu và chỉ nếu một đợt tìm kiếm độ sâu đầu tiên của G không cho ra cạnh lùi nào.

Chứng minh $\Rightarrow$: Giả sử rằng có một cạnh lùi (u, v). Thì, đỉnh v là một tiền bối của đỉnh u trong rừng độ sâu đầu tiên. Như vậy, ta có một lộ trình từ v đến u trong G, và cạnh lùi (u, v) hoàn tất một chu trình.



Hình 23.8 Một dag cho kiểu sắp xếp tôpô.

$\Leftarrow$: Giả sử G chứa một chu trình c. Ta chứng tỏ một đợt tìm kiếm độ sâu đầu tiên của G cho ra một cạnh lùi. Cho v là đỉnh đầu tiên được khám phá trong c, và cho (u, v) là cạnh đứng trước trong c. Vào thời gian $d[v]$, ta có một lộ trình các đỉnh trắng từ v sang u. Theo định lý lộ trình trắng, đỉnh u trở thành một hậu duệ của v trong rừng độ sâu đầu tiên. Do đó, (u, v) là một cạnh lùi.

Định lý 23.11

TOPOLOGICAL-SORT(G) tạo một đợt sắp xếp tôpô của một đồ thị phi chu trình có hướng G.

Chứng minh Giả sử DFS chạy trên một dag đã cho $G = (V, E)$ để xác định các thời gian hoàn tất cho các đỉnh của nó. Ta chỉ cần chứng tỏ với

một cặp các đỉnh riêng biệt $u, v \in V$ bất kỳ, nếu có một cạnh trong G từ u đến v, thì $f[v] < f[u]$. Xét bất kỳ cạnh (u, v) nào được DFS(G) khảo sát. Khi cạnh này được khảo sát, v không thể là xám, bởi v sẽ là một tiền bối của u và (u, v) sẽ là một cạnh lùi, mâu thuẫn với Bổ đề 23.10. Do đó, v phải là trắng hoặc đen. Nếu v là trắng, nó trở thành một hậu duệ của u, và do đó $f[v] < f[u]$. Nếu v là đen, thì $f[v] < f[u]$. Như vậy, với một cạnh (u, v) trong dag, ta có $f[v] < f[u]$, đồng thời chứng minh định lý.

Bài tập

23.4-1

Nêu cách sắp xếp thứ tự các đỉnh do TOPOLOGICAL-SORT tạo ra khi nó chạy trên dag của Hình 23.8.

23.4-2

Có nhiều cách sắp xếp khác nhau đối với các đỉnh của một đồ thị có hướng G là các kiểu sắp xếp tôpô của G. TOPOLOGICAL-SORT tạo ra cách sắp xếp thứ tự nghịch đảo với các thời gian hoàn tất độ sâu đầu tiên. Chứng tỏ không phải tất cả các đợt sắp xếp tôpô đều có thể được tạo ra theo cách này: ở đó tồn tại một đồ thị G sao cho TOPOLOGICAL-SORT không thể tạo ra một trong các đợt sắp xếp tôpô của G, bất kể gán cho G cấu trúc danh sách kề nào. Cũng chứng tỏ ở đó tồn tại một đồ thị mà hai phép biểu diễn danh sách kề riêng biệt đều cho ra cùng kiểu sắp xếp tôpô.

23.4-3

Nêu một thuật toán xác định một đồ thị không có hướng đã cho $G = (V, E)$ có chứa một chu trình hay không. Thuật toán của bạn phải chạy trong $O(V)$ thời gian, độc lập với $|E|$.

23.4-4

Chứng minh hoặc bác bỏ: Nếu một đồ thị có hướng G chứa các chu trình, thì TOPOLOGICAL-SORT(G) tạo ra một cách sắp xếp đỉnh giảm thiểu số lượng các cạnh "tồi" không nhất quán với cách sắp xếp thứ tự được tạo ra.

23.4-5

Một cách khác để tiến hành sắp xếp tôpô trên một đồ thị phi chu

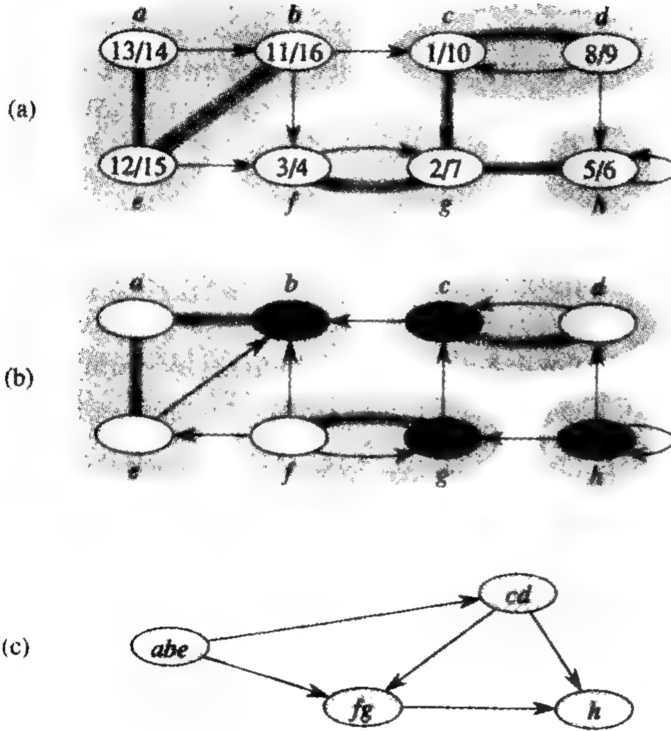
trình có hướng $G = (V, E)$ đó là liên tục tìm một đỉnh của độ vào 0, kết xuất nó, gỡ bỏ nó cùng tất cả các cạnh ra củanó ra khỏi đồ thị. Giải thích cách thực thi ý tưởng này sao cho nó chạy trong thời gian $O(V + E)$. Điều gì xảy ra với thuật toán này nếu G có các chu trình?

23.5 Các thành phần liên thông mạnh

Giờ đây ta xét một ứng dụng cổ điển của thuật toán tìm kiếm độ sâu đầu tiên: phân tích một đồ thị có hướng thành các thành phần liên thông mạnh của nó. Đoạn này sẽ trình bày cách thực hiện kiểu phân tích này bằng hai thuật toán tìm kiếm độ sâu đầu tiên. Nhiều thuật toán làm việc với đồ thị có hướng bắt đầu bằng một kiểu phân tích như vậy; cách tiếp cận này thường cho phép chia bài toán ban đầu thành các bài toán con, mỗi bài dành cho một thành phần liên thông mạnh. Tiến trình tổ hợp các giải pháp cho các bài toán con sẽ theo cấu trúc các tuyến nối giữa các thành phần liên thông mạnh; có thể biểu diễn cấu trúc này bằng một đồ thị có tên là đồ thị "thành phần," được định nghĩa trong Bài tập 23.5-4.

Hãy nhớ lại trong Chương 5, một thành phần liên thông mạnh của một đồ thị có hướng $G = (V, E)$ là một tập hợp cực đại các đỉnh $U \subseteq V$ sao cho với mọi cặp các đỉnh u và v trong U, ta có cả $u \rightsquigarrow v$ lẫn $v \rightsquigarrow u$; nghĩa là, các đỉnh u và v là khả đụng với nhau. Hình 23.9 có nêu một ví dụ.

Thuật toán của chúng ta để tìm các thành phần liên thông mạnh của một đồ thị $G = (V, E)$ sử dụng phép chuyển vị của G, được định nghĩa trong Bài tập 23.1-3 là đồ thị $G^T = (V, E^T)$, ở đó $E^T = \{(u, v) : (v, u) \in E\}$. Nghĩa là, E^T bao gồm các cạnh của G mà các hướng của chúng được đảo ngược. Cho một phép biểu diễn danh sách kề của G, thời gian để tạo G^T là $O(V + E)$. Cần lưu ý là G và G^T có chính xác cùng các thành phần liên thông mạnh: *u và v* là khả đụng với nhau trong G nếu và chỉ nếu chúng khả đụng với nhau trong G^T. Hình 23.9(b) nêu phép chuyển vị của đồ thị trong Hình 23.9(a), với các thành phần liên thông mạnh được tô bóng.



Hình 23.9 (a) Một đồ thị có hướng G. Các thành phần liên thông mạnh của G được nêu dưới dạng các vùng tô bóng. Mỗi đỉnh được gán nhãn với các thời gian khám phá và hoàn tất của nó. Các cạnh cây được tô bóng. **(b)** Đồ thị G^T, chuyển vị của G. Cây độ sâu đầu tiên đã tính toán trong dòng 3 của STRONGLY-CONNECTED-COMPONENTS được nêu, với các cạnh cây được tô bóng. Mỗi thành phần liên thông mạnh tương ứng với một cây độ sâu đầu tiên. Các đỉnh b, c, g, và h, được tô bóng đậm, là tổ tiên của mọi đỉnh trong thành phần liên thông mạnh của chúng; các đỉnh này cũng là các gốc của các cây độ sâu-đầu tiên mà đợt tìm kiếm độ sâu đầu tiên của G^T tạo ra. **(c)** Thành phần đồ thị G^{scc} phi chu trình có được nhờ rút gọn từng thành phần liên thông mạnh của G thành một đỉnh đơn lẻ.

Thuật toán thời gian tuyến tính dưới đây (tức, $\Theta(V + E)$ thời gian) sẽ tính các thành phần liên thông mạnh của một đồ thị có hướng $G = (V, E)$ dùng hai đợt tìm kiếm độ sâu đầu tiên, một trên G và một trên G^T.

STRONGLY-CONNECTED-COMPONENTS (G)

1 gọi DFS(G) để tính toán các thời gian hoàn tất $f[u]$ với mỗi đỉnh u

2 tính toán G^T

3 gọi DFS(G^T), nhưng trong vòng lặp chính của DFS, xét các đỉnh theo thứ tự giảm $f[u]$ (như đã tính toán trong dòng 1)

4 kết xuất các đỉnh của mỗi cây trong rừng độ sâu đầu tiên của bước

3 như một thành phần liên thông mạnh tách biệt

Thuật toán tìm đơn giản này dường như chẳng dính dáng gì với các thành phần liên thông mạnh. Trong phần còn lại của đoạn này, ta vén màn bí mật của thiết kế và chứng minh tính đúng đắn củanó. Ta bắt đầu bằng hai nhận xét hữu ích.

Bổ đề 23.12

Nếu hai đỉnh nằm trong cùng thành phần liên thông mạnh, thì không bao giờ có lộ trình nào giữa chúng rời thành phần liên thông mạnh.

Chứng minh Cho *u* và *v* là hai đỉnh trong cùng thành phần liên thông mạnh. Qua phần định nghĩa của thành phần liên thông mạnh, ta có các lộ trình từ *u* đến *v* và từ *v* đến *u*. Cho đỉnh *w* nằm trên một lộ trình $u \leadsto w \leadsto v$, sao cho *w* khả dụng từ *u*. Hơn nữa, do có một lộ trình $v \leadsto u$, ta biết *u* khả dụng từ *w* theo lộ trình $w \leadsto v \leadsto u$. Do đó, *u* và *w* nằm trong cùng thành phần liên thông mạnh. Bởi *w* được chọn một cách tùy ý, định lý được chứng minh.

Định lý 23.13

Trong mọi đợt tìm kiếm độ sâu đầu tiên, tất cả các đỉnh trong cùng thành phần liên thông mạnh đều được đặt trong cùng cây độ sâu đầu tiên.

Chứng minh Trong số các đỉnh trong thành phần liên thông mạnh, ta cho *r* là đỉnh đầu tiên được khám phá. Do *r* là đầu tiên, nên các đỉnh khác trong thành phần liên thông mạnh là trắng vào lúc nó được khám phá. Ta có các lộ trình từ *r* đến mọi đỉnh khác trong thành phần liên thông mạnh; do các lộ trình này không bao giờ rời thành phần liên thông mạnh (theo Bổ đề 23.12), tất cả các đỉnh trên chúng là trắng. Như vậy, theo định lý lộ trình trắng, mọi đỉnh trong thành phần liên thông mạnh trở thành một hậu duệ của *r* trong cây độ sâu đầu tiên.

Trong phần còn lại của đoạn này, các hệ ký hiệu $d[u]$ và $f[u]$ ám chỉ các thời gian khám phá và hoàn tất như đã được tính toán bởi đợt tìm kiếm độ sâu đầu tiên lần đầu trong dòng 1 của STRONGLY-CONNECTED-COMPONENTS. Cũng vậy, hệ ký hiệu $u \leadsto v$ ám chỉ sự hiện diện của một lộ trình trong *G*, chứ không phải trong G^T.

Để chứng minh STRONGLY-CONNECTED-COMPONENTS là đúng đắn, ta giới thiệu khái niệm về ***tổ tiên*** $\phi(u)$ của một đỉnh *u*, là đỉnh *w* khả dụng từ *u* được hoàn tất cuối cùng trong đợt tìm kiếm độ sâu đầu tiên của dòng 1. Nói cách khác,

$\phi(u)$ = đỉnh *w* đó sao cho $u \leadsto w$ và $f[w]$ được cực đại hóa.

Lưu ý, $\phi(u) = u$ là khả dĩ bởi u là khả dụng từ chính nó, và do đó

$$f[u] \le f[\phi(u)] \cdot \qquad (23.2)$$

Cũng có thể chứng tỏ $\phi(\phi(u)) = \phi(u)$, theo biện luận dưới đây. Với mọi đỉnh $u, v \in V$,

$$u, v \text{ hàm ý } f[\phi(v)] \le f[\phi(u)], \qquad (23.3)$$

bởi $\{w : v \rightsquigarrow w\} \subseteq \{w : u \rightsquigarrow w\}$ và tổ tiên có thời gian kết thúc cực đại của tất cả các đỉnh khả dụng. Bởi $\phi(u)$ là khả dụng từ u, công thức (23.3) hàm ý $f[\phi(\phi(u))] \le f[\phi(u)]$. Ta cũng có $f[\phi(u)] \le f[\phi(\phi(u))]$, theo bất đẳng thức (23.2). Như vậy, $f[\phi(\phi(u))] = f[\phi(u)]$, và do đó ta có $\phi(\phi(u)) = \phi(u)$, bởi hai đỉnh hoàn tất tại cùng thời gian thực tế là cùng một đỉnh.

Như sẽ thấy, mọi thành phần liên thông mạnh có một đỉnh là tổ tiên của mọi đỉnh trong thành phần liên thông mạnh; tổ tiên này là một "đỉnh đại diện" cho thành phần liên thông mạnh. Trong đợt tìm kiếm độ sâu đầu tiên của G, nó là đỉnh đầu tiên của thành phần liên thông mạnh được khám phá, và nó là đỉnh cuối của thành phần liên thông mạnh được hoàn tất. Trong đợt tìm kiếm độ sâu đầu tiên của G^T, nó là gốc của một cây độ sâu đầu tiên. Giờ đây ta chứng minh các tính chất này.

Định lý đầu tiên xác minh cách gọi $\phi(u)$ là một "tổ tiên" của u.

Định lý 23.14

Trong một đồ thị có hướng $G = (V, E)$, tổ tiên $\phi(u)$ của một đỉnh $u \in V$ trong mọi đợt tìm kiếm độ sâu đầu tiên của G chính là một tiền bối của u.

Chứng minh Nếu $\phi(u) = u$, định lý đương nhiên là đúng. Nếu $\phi(u) \ne u$, ta hãy xét các màu của các đỉnh vào thời gian $d[u]$. Nếu $\phi(u)$ là đen, thì $f[\phi(u)] < f[u]$, mâu thuẫn với bất đẳng thức (23.2). Nếu $\phi(u)$ là xám, thì nó là một tiền bối của u, và định lý được chứng minh.

Như vậy ta chỉ cần chứng minh $\phi(u)$ không trắng. Có hai trường hợp, theo các màu của các đỉnh trung gian, nếu có, trên lộ trình từ u đến $\phi(u)$.

1. Nếu mọi đỉnh trung gian là trắng, thì $\phi(u)$ trở thành một hậu duệ của u, theo định lý lộ trình trắng. Nhưng như vậy thì $f[\phi(u)] < f[u]$, mâu thuẫn với bất đẳng thức (23.2).

2. Nếu có một đỉnh trung gian không trắng, cho t là đỉnh không trắng cuối cùng trên lộ trình từ u đến $\phi(u)$. Vậy, t phải là xám, bởi không bao giờ có một cạnh từ một đỉnh đen đến một đỉnh trắng, và phần tử kế vị của t là trắng. Song vậy thì ta có một lộ trình các đỉnh trắng từ t đến $\phi(u)$,

và do đó $\phi(u)$ là một hậu duệ của t theo định lý lộ trình trắng. Điều này hàm ý $f[t] > f[\phi(u)]$, mâu thuẫn với sự chọn lựa của chúng ta về $\phi(u)$, bởi có một lộ trình từ u đến t.

Hệ luận 23.15

Trong mọi đợt tìm kiếm độ sâu đầu tiên của một đồ thị có hướng $G = (V, E)$, các đỉnh u và $\phi(u)$, với tất cả $u \in V$, nằm trong cùng thành phần liên thông mạnh.

Chứng minh Ta có $u \rightsquigarrow \phi(u)$, theo định nghĩa về tổ tiên, và $\phi(u) \rightsquigarrow u$, bởi $\phi(u)$ là một tiền bối của u.

Định lý dưới đây cho ra một kết quả mạnh hơn liên kết các tổ tiên với các thành phần liên thông mạnh.

Định lý 23.16

Trong một đồ thị có hướng $G = (V, E)$, hai đỉnh $u, v \in V$ nằm trong cùng thành phần liên thông mạnh nếu và chỉ nếu chúng có cùng tổ tiên trong một đợt tìm kiếm độ sâu đầu tiên của G.

Chứng minh $\Rightarrow$: Giả sử u *và* v nằm trong cùng thành phần liên thông mạnh. Mọi đỉnh khả đụng từ u sẽ khả đụng từ v và ngược lại, bởi ta có các lộ trình trong cả hai hướng giữa u và v. Theo định nghĩa về tổ tiên, ta kết luận $\phi(u) = \phi(v)$.

$\Leftarrow$: Giả sử $\phi(u) = \phi(v)$. Theo Hệ luận 23.15, u nằm trong cùng thành phần liên thông mạnh như $\phi(u)$, và v nằm trong cùng thành phần liên thông mạnh như $\phi(v)$. Do đó, u và v nằm trong cùng thành phần liên thông mạnh.

Với Định lý 23.16 trong tay, cấu trúc của thuật toán STRONGLY-CONNECTED-COMPONENTS có thể sáng tỏ hơn. Các thành phần liên thông mạnh là những tập hợp các đỉnh có cùng tổ tiên. Hơn nữa, theo Định lý 23.14 và định lý dấu ngoặc đơn (Định lý 23.6), trong đợt tìm kiếm độ sâu đầu tiên trong dòng 1 của STRONGLY-CONNECTED-COMPONENTS, một tổ tiên vừa là đỉnh đầu tiên đã khám phá vừa là đỉnh cuối hoàn tất trong thành phần liên thông mạnh củanó.

Để hiểu tại sao ta chạy đợt tìm kiếm độ sâu đầu tiên trong dòng 3 của STRONGLY-CONNECTED-COMPONENTS trên G^T, hãy xét đỉnh r có thời gian kết thúc lớn nhất mà đợt tìm kiếm độ sâu đầu tiên trong dòng 1 đã tính toán. Theo định nghĩa về tổ tiên, đỉnh r phải là một tổ tiên, bởi nó là tổ tiên của chính nó: nó có thể tự đụng, và không có đỉnh nào trong đồ thị có một thời gian kết thúc cao hơn. Đâu là các đỉnh khác trong thành phần liên thông mạnh của r? Chúng là những đỉnh có r làm một tổ tiên—những đỉnh có thể đụng r nhưng không thể đụng bất kỳ

đỉnh nào có một thời gian kết thúc lớn hơn $f[r]$. Nhưng thời gian kết thúc của r là cực đại của một đỉnh bất kỳ trong G; như vậy, thành phần liên thông mạnh một r đơn giản bao gồm những đỉnh có thể đụng r. Một cách tương đương, *thành phần liên thông mạnh của r bao gồm những đỉnh mà r có thể đụng trong* G^T. Như vậy, đợt tìm kiếm độ sâu đầu tiên trong dòng 3 định danh tất cả các đỉnh trong thành phần liên thông mạnh của r và tô đen chúng. (Một đợt tìm kiếm độ rộng đầu tiên, hoặc *bất kỳ* đợt tìm kiếm các đỉnh khả dụng, có thể định danh tập hợp này dễ dàng không kém.)

Sau khi đợt tìm kiếm độ sâu đầu tiên trong dòng 3 hoàn tất việc định danh thành phần liên thông mạnh của r, nó bắt đầu tại đỉnh r' với thời gian kết thúc lớn nhất của một đỉnh bất kỳ không nằm trong thành phần liên thông mạnh của r. Đỉnh r' phải là tổ tiên của riêng nó, bởi nó không thể đụng bất kỳ đỉnh nào có một thời gian kết thúc cao hơn (bằng không, nó sẽ được gộp trong thành phần liên thông mạnh của r). Theo biện luận tương tự, bất kỳ đỉnh nào có thể đụng r' chưa được tô đen đều phải nằm trong thành phần liên thông mạnh của r'. Như vậy, khi đợt tìm kiếm độ sâu đầu tiên trong dòng 3 tiếp tục, nó định danh và tô đen mọi đỉnh trong thành phần liên thông mạnh của r' bằng cách tìm từ r' trong G^T.

Như vậy, đợt tìm kiếm độ sâu đầu tiên trong dòng 3 sẽ lần lượt "lột" từng thành phần liên thông mạnh. Mỗi thành phần được định danh trong dòng 7 của DFS bằng một lệnh gọi DFS-VISIT có tổ tiên của thành phần đó làm đối số. Các lệnh gọi đệ quy trong DFS-VISIT chung cuộc sẽ tô đen từng đỉnh trong thành phần. Khi DFS-VISIT trở về DFS, nguyên cả thành phần đã được tô đen và "lột." Sau đó, DFS tìm đỉnh có thời gian kết thúc cực đại giữa các đỉnh chưa được tô đen; đỉnh này là tổ tiên của một thành phần khác, và tiến trình tiếp tục.

Định lý dưới đây hình thức hóa đối số này.

Định lý 23.17

STRONGLY-CONNECTED-COMPONENTS(G) tính toán đúng đắn các thành phần liên thông mạnh của một đồ thị có hướng G.

Chứng minh Bằng phương pháp quy nạp trên số lượng các cây độ sâu đầu tiên tìm thấy trong đợt tìm kiếm độ sâu đầu tiên của G^T, ta chứng tỏ các đỉnh của mỗi cây hình thành một thành phần liên thông mạnh. Mỗi bước của đối số quy nạp sẽ chứng minh rằng một cây được hình thành trong đợt tìm kiếm độ sâu đầu tiên của G^T là một thành phần kiên thông mạnh, giả định tất cả các cây trước đó được tạo đều là các thành phần liên thông mạnh. Cơ sở cho phương pháp quy nạp chẳng có

gì đáng nói, bởi với cây đầu tiên được tạo không có các cây trước đó, và do đó giả thiết này hiển nhiên là đúng.

Xét một cây độ sâu đầu tiên T có gốc r được tạo trong đợt tìm kiếm độ sâu đầu của G^T. Cho $C(r)$ thể hiện tập hợp các đỉnh có tổ tiên r:

$C(r) = \{v \in V : \phi(v) = r\}$.

Giờ đây ta chứng minh một đỉnh u được đặt trong T nếu và chỉ nếu $u \in C(r)$.

$\Leftarrow$: Định lý 23.13 hàm ý mọi đỉnh trong $C(r)$ đều kết thúc trong cùng cây độ sâu đầu tiên. Bởi $r \in C(r)$ và r là gốc của T, nên mọi thành phần của $C(r)$ kết thúc trong T.

$\Rightarrow$: Ta chứng tỏ mọi đỉnh w sao cho $f[\phi(w)] > f[r]$ hoặc $f[\phi(w)] < f[r]$ đều không được đặt trong T, bằng cách xem xét hai trường hợp này tách biệt. Theo phương pháp quy nạp trên số lượng các cây tìm thấy, bất kỳ đỉnh w nào sao cho $f[\phi(w)] > f[r]$ không được đặt trong cây T, bởi vào thời gian mà r được chọn w tất đã được đặt trong cây có gốc $\phi(w)$. Bất kỳ đỉnh w nào sao cho $f[\phi(w)] < f[r]$ đều không thể được đặt trong T, bởi cách đặt như vậy hàm ý $w \leadsto r$; như vậy, theo công thức (23.3) và tính chất $r = \phi(r)$, ta được $f[\phi(w)] \geq f[\phi(r)] = f[r]$, mâu thuẫn với $f[\phi(w)] < f[r]$.

Do đó, T chỉ chứa những đỉnh u mà $\phi(u) = r$. Nghĩa là, T chính xác bằng với thành phần liên thông mạnh $C(r)$, hoàn tất chứng minh quy nạp.

Bài tập

23.5-1

Số lượng các thành phần liên thông mạnh của một đồ thị có thể thay đổi như thế nào nếu ta bổ sung một cạnh mới?

23.5-2

Nêu cách làm việc của thủ tục STRONGLY-CONNECTED-COMPONENTS trên đồ thị của Hình 23.6. Cụ thể, nêu các thời gian kết thúc đã tính toán trong dòng 1 và rừng được tạo trong dòng 3. Giả sử vòng lặp của các dòng 5-7 của DFS xét các đỉnh theo thứ tự abc và các danh sách kề theo thứ tự abc.

23.5-3

Giáo sư Deaver cho rằng thuật toán của các thành phần liên thông mạnh có thể được đơn giản hóa bằng cách dùng đồ thị ban đầu (thay vì chuyển vị) trong đợt tìm kiếm độ sâu đầu tiên thứ hai và quét các đỉnh

theo thứ tự các thời gian kết thúc *tăng dần*. Giáo sư có đúng không?

23.5-4

Ta thể hiện ***đồ thị thành phần*** của $G = (V, E)$ bằng $G^{scc} = (V^{scc}, E^{scc})$, ở đó V^{scc} chứa một đỉnh cho mỗi thành phần liên thông mạnh của G và E^{scc} chứa cạnh (u, v) nếu có một cạnh có hướng từ một đỉnh trong thành phần liên thông mạnh của G tương ứng với u đến một đỉnh trong thành phần liên thông mạnh của G tương ứng với v. Hình 23.9(c) nêu một ví dụ. Chứng minh G^{scc} là một đồ thị phi chu trình có hướng [dag].

23.5-5

Nêu một thuật toán $O(V + E)$ thời gian để tính toán đồ thị thành phần của một đồ thị có hướng $G = (V, E)$. Bảo đảm có tối đa một cạnh giữa hai đỉnh trong đồ thị thành phần mà thuật toán của bạn tạo ra.

23.5-6

Cho một đồ thị có hướng $G = (V, E)$, giải thích cách tạo một đồ thị G' $= (V, E')$ khác sao cho (a) G' có cùng các thành phần liên thông mạnh như G, (b) G' có cùng đồ thị thành phần như G, và (c) E' càng nhỏ càng tốt. Mô tả một thuật toán nhanh để tính toán G'.

23.5-7

Một đồ thị có hướng $G = (V, E)$ được xem là ***bán liên thông*** [semiconnected] nếu, với bất kỳ hai đỉnh $u, v \in V$, ta có $u \rightsquigarrow v$ hoặc $v \rightsquigarrow u$. Nêu một thuật toán hiệu quả để xác định G có bán liên thông hay không. Chứng minh thuật toán của bạn là đúng, và phân tích thời gian thực hiện của nó.

Các Bài toán

23-1 Phân loại các cạnh bằng thuật toán tìm kiếm độ rộng đầu tiên

Một rừng độ sâu đầu tiên phân loại các cạnh của một đồ thị thành các cạnh cây, lùi, tới, và chéo. Một cây độ rộng đầu tiên cũng có thể được dùng để phân loại các cạnh khả dụng từ nguồn của đợt tìm kiếm thành bốn phạm trù giống nhau.

a. Chứng minh trong một đợt tìm kiếm độ rộng đầu tiên của một đồ thị không có hướng, các tính chất dưới đây sẽ đứng vững:

1. Không có các cạnh lùi và các cạnh tới.

2. Với mỗi cạnh cây (u, v), ta có $d[v] = d[u] + 1$.

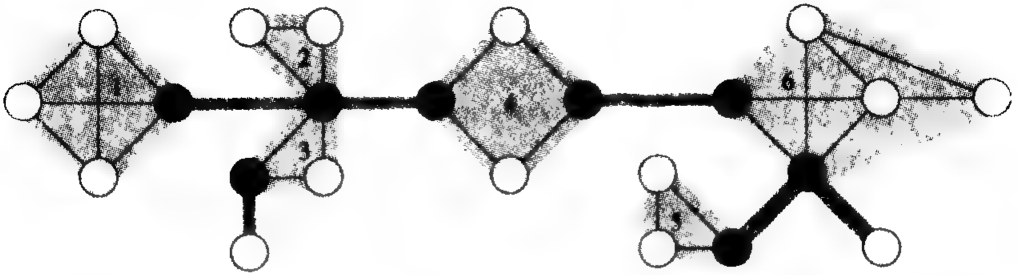
3. Với mỗi cạnh chéo (u, v), ta có $d[v] = d[u]$ hoặc $d[v] = d[u] + 1$.

b. Chứng minh trong một đợt tìm kiếm độ rộng đầu tiên của một đồ thị có hướng, các tính chất dưới đây đứng vững:

1. Không có các cạnh tới.
2. Với mỗi cạnh cây (u, v), ta có $d[v] = d[u] + 1$.
3. Với mỗi cạnh chéo (u, v), ta có $d[v] \le d[u] + 1$.
4. Với mỗi cạnh lùi (u, v), ta có $0 \le d[v] \le d[u]$.

23-2 Các điểm khớp nối, các cầu nối, và các thành phần lưỡng thông

Cho $G = (V, E)$ là một đồ thị không có hướng, liên thông. Một ***điểm khớp nối*** [articulation point] của G là một đỉnh mà việc gỡ bỏ nó sẽ làm gián đoạn G. Một ***cầu nối*** [bridge] của G là một cạnh mà việc gỡ bỏ nó sẽ làm gián đoạn G. Một ***thành phần lưỡng thông*** [biconnected component] của G là một tập hợp cực đại các cạnh sao cho hai cạnh bất kỳ trong tập hợp nằm trên một chu trình đơn giản chung. Hình 23.10 minh họa các định nghĩa này. Ta có thể xác định các điểm khớp nối, các cầu nối, và các thành phần lưỡng thông bằng thuật toán tìm kiếm độ sâu đầu tiên. Cho $G_\pi = (V, E_\pi)$ là một cây độ sâu đầu tiên của G.



Hình 23.10 Các điểm khớp nối, các cầu nối, và các thành phần lưỡng thông của một đồ thị không có hướng, liên thông, để dùng trong Bài toán 23-2. Các điểm khớp nối là các đỉnh tô bóng đậm, các cầu nối là các cạnh được tô bóng đậm, và các thành phần lưỡng thông là các cạnh trong các vùng tô bóng, với một kiểu đánh số *bcc* đã nêu.

a. Chứng minh gốc của G_π là một điểm khớp nối của G nếu và chỉ nếu nó có ít nhất hai con trong G_π.

b. Cho v là một đỉnh phi gốc trong G_π. Chứng minh v là một điểm khớp nối của G nếu và chỉ nếu không có cạnh lùi (u, w) nào sao cho trong G_π, u là một hậu duệ của v và w là một tiền bối riêng của v.

c. Cho

$$low[v] = \min \begin{cases} d[v], \\ \{d[w] : (u, w) \text{ là một cạnh lùi} \\ \quad \text{với một hậu duệ } u \text{ của } v\}. \end{cases}$$

Nêu cách tính $low[v]$ với tất cả các đỉnh $v \in V$ trong $O(E)$ thời gian.

d. Nêu cách tính tất cả các điểm khớp nối trong $O(E)$ thời gian.

e. Chứng minh một cạnh của G là một cầu nối nếu và chỉ nếu nó không nằm trên bất kỳ chu trình đơn giản nào của G.

f. Nêu cách tính tất cả các cầu nối của G trong $O(E)$ thời gian.

g. Chứng minh các thành phần lưỡng thông của G phân hoạch các cạnh phi cầu của G.

h. Nêu một thuật toán $O(E)$ thời gian gán nhãn mỗi cạnh e của G với một số nguyên dương $bcc[e]$ sao cho $bcc[e] = bcc[e']$ nếu và chỉ nếu e và e' nằm trong cùng thành phần lưỡng thông.

23-3 Tua Euler

Một ***tua Euler*** của một đồ thị có hướng, liên thông $G = (V, E)$ là một chu trình băng ngang mỗi cạnh của G chính xác một lần, mặc dù nó có thể ghé một đỉnh nhiều lần.

a. Chứng tỏ G có một tua Euler nếu và chỉ nếu

độ vào(v) = độ ra(v)

với mỗi đỉnh $v \in V$.

b. Mô tả một thuật toán $O(E)$ thời gian để tìm một tua Euler của G nếu tồn tại một tua như vậy. (*Mách nước:* Hợp nhất các chu trình cạnh rời nhau.)

Ghi chú Chương

Even [65] và Tarjan [188] là các tham chiếu tuyệt vời về các thuật toán đồ thị.

Thuật toán tìm kiếm độ rộng đầu tiên đã được Moore [150] khám phá trong ngữ cảnh tìm các lộ trình qua các mê cung. Lee [134] đã độc lập khám phá cùng thuật toán trong ngữ cảnh định tuyến các dây dẫn trên các bo mạch.

Hopcroft và Tarjan [102] đã ủng hộ cách dùng phép biểu diễn danh sách kề so với phép biểu diễn ma trận kề với đồ thị thưa và là những người đầu tiên nhận ra tầm quan trọng về thuật toán của kỹ thuật tìm

kiếm độ sâu đầu tiên. Tìm kiếm độ sâu đầu tiên đã được dùng rộng rãi vào cuối những năm 1950, nhất là trong các chương trình trí tuệ nhân tạo.

Tarjan [185] đã cung cấp một thuật toán thời gian tuyến tính để tìm các thành phần liên thông mạnh. Thuật toán về các thành phần liên thông mạnh trong Đoạn 23.5 được thích ứng từ Aho, Hopcroft, và Ullman [5], là những người đã quy nó cho S. *R*. Kosaraju và M. Sharir. Knuth [121] là người đầu tiên cung cấp một thuật toán thời gian tuyến tính dành cho kỹ thuật sắp xếp tôpô.

24 Các Cây Tỏa Nhánh Tối Thiểu

Trong khâu thiết kế hệ mạch điện tử, ta thường phải tạo các kim [pins] của vài thành phần tương đương về điện tử bằng cách đấu dây chúng với nhau. Để tương kết một tập hợp n kim, ta có thể dùng một cách bố trí $n - 1$ dây dẫn, mỗi dây nối hai kim. Trong số tất cả các kiểu bố trí như vậy, kiểu sử dụng lượng dây dẫn ít nhất thường thỏa đáng nhất.

Ta có thể mô hình hóa bài toán đấu dây này với một đồ thị không hướng, liên thông $G = (V, E)$, ở đó V là tập hợp các kim, E là tập hợp các tuyến tương kết khả dĩ giữa các cặp kim, và với mỗi cạnh $(u, v) \in E$, ta có một trọng số $w(u, v)$ chỉ định mức hao phí (lượng dây dẫn cần thiết) để nối u và v. Như vậy, ta muốn tìm một tập hợp con phi chu trình $T \subseteq E$ liên thông tất cả các đỉnh và tổng trọng số của chúng

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

được giảm thiểu. Bởi T là phi chu trình và liên thông tất cả các đỉnh, nên nó phải hình thành một cây, mà ta gọi là một ***cây tỏa nhánh*** bởi nó "tỏa rộng" đồ thị G. Ta gọi bài toán xác định cây T là ***bài toán cây tỏa nhánh cực tiểu*** [minimum-spanning-tree problem].[1] Hình 24.1 có nêu một ví dụ của một đồ thị liên thông và cây tỏa nhánh cực tiểu của nó.

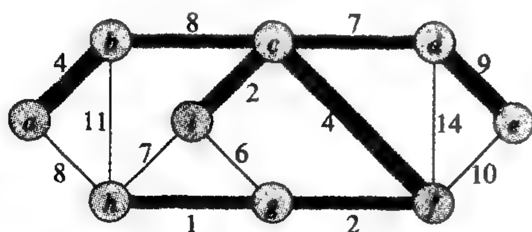
Trong chương này, ta sẽ xét hai thuật toán để giải quyết bài toán cây tỏa nhánh cực tiểu: thuật toán Kruskal và thuật toán Prim. Có thể dễ dàng tạo chúng để chạy trong thời gian $O(E \lg V)$ bằng các đống nhị phân bình thường. Nhờ dùng các đống Fibonacci, thuật toán Prim có thể tăng tốc để chạy trong thời gian $O(E + V \lg V)$, là một cải tiến nếu $|V|$ nhỏ hơn nhiều so với $|E|$.

Hai thuật toán này cũng minh họa một heuristic để tối ưu hóa có tên chiến lược "tham" [greedy]. Tại mỗi bước của một thuật toán, ta phải thực hiện một trong số vài chọn lựa khả dĩ. Chiến lược tham chủ trương

---

[1] Câu "cây tỏa nhánh cực tiểu" là một dạng viết gọn của câu "cây tỏa nhánh có trọng số cực tiểu." Ví dụ, ta không giảm thiểu số lượng các cạnh trong T, bởi theo Định lý 5.2, tất cả các cây tỏa nhánh có chính xác $|V| - 1$ cạnh.

thực hiện sự chọn lựa tốt nhất vào lúc đó. Một chiến lược như vậy thường không được bảo đảm là sẽ tìm ra các giải pháp tối ưu toàn cục cho các bài toán. Tuy nhiên, với bài toán cây tỏa nhánh cực tiểu, ta có thể chứng minh một số chiến lược tham nhất định sẽ cho ra một cây tỏa nhánh có trọng số cực tiểu. Các chiến lược tham đã được mô tả chi tiết trong Chương 17. Mặc dù có thể đọc chương này một cách độc lập với Chương 17, song các phương pháp tham được trình bày ở đây là một ứng dụng cổ điển của các khái niệm lý thuyết đã giới thiệu trong chương đó.



Hình 24.1 Một cây tỏa nhánh cực tiểu cho một đồ thị liên thông. Các trọng số trên các cạnh được nêu, và các cạnh trong một cây tỏa nhánh cực tiểu được tô bóng. Tổng trọng số của cây được nêu là 37. Cây không phải là duy nhất: việc gỡ bỏ cạnh (b, c) và thay nó bằng cạnh (a, h) sẽ cho ra một cây tỏa nhánh khác có trọng số 37.

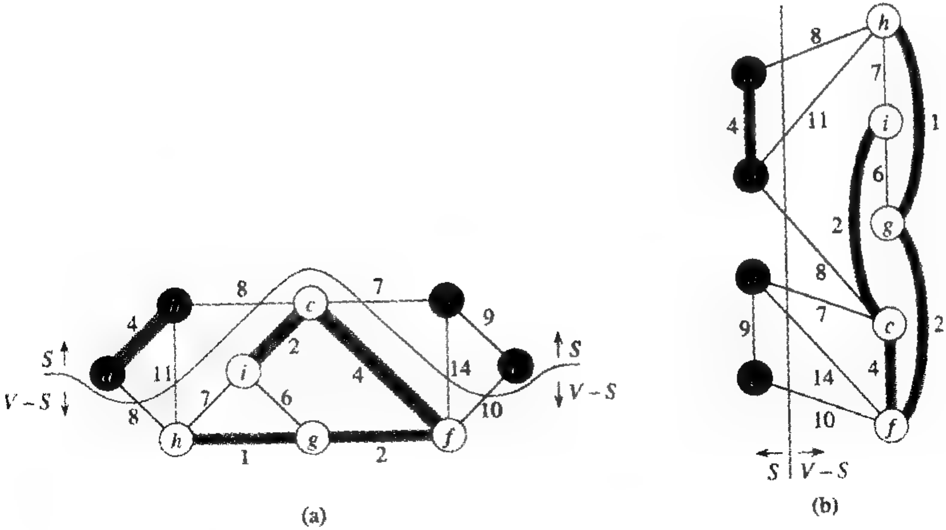
Đoạn 24.1 giới thiệu một thuật toán cây tỏa nhánh cực tiểu "chung" tăng trưởng một cây tỏa nhánh bằng cách lần lượt bổ sung từng một cạnh một. Đoạn 24.2 đưa ra hai cách để thực thi thuật toán chung. Thuật toán đầu tiên, của Kruskal, tương tự như thuật toán các thành phần liên thông trong Đoạn 22.1. Thuật toán thứ hai, của Prim, tương tự như thuật toán các lộ trình ngắn nhất của Dijkstra (Đoạn 25.2).

24.1 Tăng trưởng một cây tỏa nhánh cực tiểu

Giả sử ta có một đồ thị không hướng, liên thông $G = (V, E)$ có một hàm trọng số $w : E \to \mathbf{R}$ và muốn tìm một cây tỏa nhánh cực tiểu cho G. Hai thuật toán mà ta xét trong chương này sử dụng cách tiếp cận tham cho bài toán, mặc dù chúng khác nhau trong cách áp dụng cách tiếp cận này.

Chiến lược tham này được chốt giữ bởi thuật toán "chung" này, tăng trưởng cây tỏa nhánh cực tiểu mỗi lần một cạnh. Thuật toán quản lý một tập hợp A luôn là một tập hợp con của một cây tỏa nhánh cực tiểu. Tại mỗi bước, một cạnh (u, v) được xác định là có thể được bổ sung vào A mà không vi phạm kiểu bất biến này, theo nghĩa là $A \cup \{(u, v)\}$ cũng

là một tập hợp con của một cây tỏa nhánh cực tiểu. Ta gọi cạnh như vậy là một ***cạnh an toàn*** của A, bởi có thể an toàn bổ sung nó vào A mà không hủy sự bất biến.



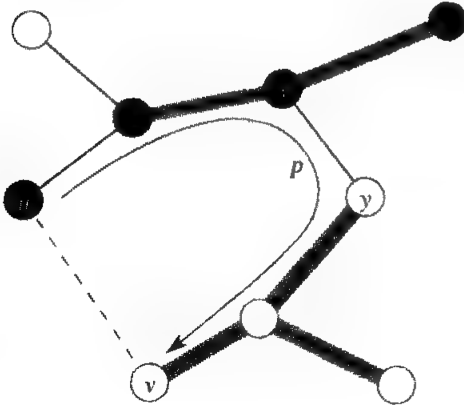
Hình 24.2 Hai cách xem một phần cắt $(S, V - S)$ của đồ thị từ Hình 24.1. **(a)** Các đỉnh trong tập hợp S được nêu ở dạng đen, và các đỉnh trong $V - S$ được nêu ở dạng trắng. Các cạnh đi qua phần cắt là những cạnh nối các đỉnh trắng với các đỉnh đen. Cạnh (d, c) là cạnh nhạt duy nhất đi qua phần cắt. Một tập hợp con A gồm các cạnh được tô bóng; lưu ý, phần cắt $(S, V - S)$ tôn trọng A, bởi không có cạnh nào của A đi qua phần cắt. **(b)** Cùng đồ thị với các đỉnh trong tập hợp S bên trái và các đỉnh trong tập hợp $V - S$ bên phải. Một cạnh đi qua phần cắt nếu nó nối một đỉnh bên trái với một đỉnh bên phải.

```
GENERIC-MST(G, w)
1  A ← ∅
2  while A không hình thành một cây tỏa nhánh
3       do tìm ra một cạnh (u, v) an toàn cho A
4           A ← A ∪ {(u, v)}
5  return A
```

Lưu ý, sau dòng 1, tập hợp A ắt thỏa sự bất biến rằng nó là một tập hợp con của một cây tỏa nhánh cực tiểu. Vòng lặp trong các dòng 2-4 duy trì sự bất biến. Do đó, khi tập hợp A được trả về trong dòng 5, nó phải là một cây tỏa nhánh cực tiểu. Tất nhiên, phần tinh tế đó là tìm một cạnh an toàn trong dòng 3. Một cạnh như vậy phải tồn tại, bởi khi dòng 3 được thi hành, sự bất biến ra lệnh có một cây tỏa nhánh T sao cho $A \subseteq T$, và nếu có một cạnh $(u, v) \in T$ sao cho $(u, v) \in A$, thì $(u,$

v) an toàn cho A.

Trong phần còn lại của đoạn này, ta đưa ra một quy tắc (Định lý 24. 1) để nhận dạng các cạnh an toàn. Đoạn kế tiếp mô tả hai thuật toán dùng quy tắc này để tìm các cạnh an toàn một cách hiệu quả.



Hình 24.3 Phần chứng minh của Định lý 24.1. Các đỉnh trong S được tô đen, và các đỉnh trong V - S là trắng. Các cạnh trong cây tỏa nhánh cực tiểu T được nêu, nhưng các cạnh trong đồ thị G không được nêu. Các cạnh trong A được tô bóng, và (u, v) là một cạnh nhạt đi qua phần cắt (S, V - S). Cạnh (x, y) là một cạnh trên lộ trình p duy nhất từ u đến v trong T. Một cây tỏa nhánh cực tiểu T' chứa (u, v) được hình thành bằng cách gỡ bỏ cạnh (x, y) ra khỏi T và bổ sung cạnh (u, v).

Trước tiên ta cần làm rõ vài định nghĩa. Một ***phần cắt*** (S, V - S) của một đồ thị không hướng $G = (V, E)$ là một phân hoạch của V. Hình 24.2 minh họa khái niệm này. Ta nói một cạnh $(u, v) \in E$ ***đi qua*** phần cắt (S, V - S) nếu một trong các điểm cuối của nó nằm trong S và điểm kia nằm trong V - S. Ta nói một phần cắt ***tôn trọng*** tập hợp A gồm các cạnh nếu không có cạnh nào trong A đi qua phần cắt. Một cạnh là một ***cạnh nhạt*** đi qua một phần cắt nếu trọng số của nó là cực tiểu của bất kỳ cạnh nào đi qua phần cắt. Lưu ý, có thể có nhiều cạnh nhạt đi qua một phần cắt trong trường hợp các mối ràng buộc. Chung hơn, ta nói rằng một cạnh là một ***cạnh nhạt*** thỏa một tính chất đã cho nếu trọng số của nó là cực tiểu của bất kỳ cạnh nào thỏa tính chất đó.

Định lý dưới đây nêu quy tắc của chúng ta để nhận dạng các cạnh an toàn.

Định lý 24.1

Cho $G = (V, E)$ là một đồ thị không hướng, liên thông với một hàm trọng số có giá trị thực w được định nghĩa trên E. Cho A là một tập hợp

con của E nằm trong vài cây tỏa nhánh cực tiểu của G, cho $(S, V\text{-}S)$ là bất kỳ phần cắt nào của G tôn trọng A, và cho (u, v) là một cạnh nhạt đi qua $(S, V - S)$. Thì, cạnh (u, v) là an toàn đối với A.

Chứng minh Cho T là một cây tỏa nhánh cực tiểu gộp A, và mặc nhận T không chứa cạnh nhạt (u, v), bởi nếu là vậy, ta đã hoàn tất. Ta sẽ kiến tạo một cây tỏa nhánh cực tiểu T' khác gộp $A \cup \{(u,v)\}$ bằng kỹ thuật cắt-và-dán, do đó chứng tỏ (u, v) là một cạnh an toàn của A.

Cạnh (u, v) hình thành một chu trình có các cạnh trên lộ trình p từ u đến v trong T, như minh họa ở Hình 24.3. Bởi u và v nằm trên các cạnh đối của phần cắt $(S, V - S)$, nên có ít nhất một cạnh trong T trên lộ trình p cũng đi qua phần cắt. Cho (x, y) là một cạnh bất kỳ như vậy. Cạnh (x, y) không nằm trong A, bởi phần cắt tôn trọng A. Do (x, y) nằm trên lộ trình duy nhất từ u đến v trong T, việc gỡ bỏ (x, y) sẽ tách T thành hai thành phần. Việc bổ sung (u, v) sẽ nối lại chúng để tạo thành một cây tỏa nhánh mới $T' = T - \{(x, y)\} \cup \{(u, v)\}$.

Kế đó, ta chứng tỏ T' là một cây tỏa nhánh cực tiểu. Bởi (u, v) là một cạnh nhạt đi qua $(S, V - S)$ và (x, y) cũng đi qua phần cắt này, nên $w(u, v) \le w(x, y)$. Do vậy,

$$\begin{aligned} w(T') &= w(T) - w(x,y) + w(u,v) \\ &\le w(T). \end{aligned}$$

Nhưng T là một cây tỏa nhánh cực tiểu, sao cho $w(T) \le w(T')$; như vậy, T' cũng phải là một cây tỏa nhánh cực tiểu.

Phần còn lại ta chỉ cần chứng tỏ (u, v) thực tế là một cạnh an toàn của A. Ta có $A \subseteq T'$, bởi $A \subseteq T$ và $(x, y) \notin A$; như vậy, $A \cup \{(u, v)\} \subseteq T'$. Kết quả, do T' là một cây tỏa nhánh cực tiểu, nên (u, v) an toàn đối với A.

Định lý 24.1 giúp ta hiểu rõ công việc của thuật toán GENERIC-MST trên một đồ thị liên thông $G = (V, E)$. Khi thuật toán tiến hành, tập hợp A luôn là phi chu trình; bằng không, một cây tỏa nhánh cực tiểu bao gồm A sẽ chứa một chu trình, là một sự mâu thuẫn. Tại một điểm bất kỳ trong khi tiến hành thuật toán, đồ thị $G_A = (V, A)$ là một rừng, và mỗi thành phần liên thông của G_A là một cây. (Một số cây có thể chỉ chứa một đỉnh, ví dụ như trường hợp khi thuật toán bắt đầu: A là trống và rừng chứa $|V|$ cây, mỗi cây cho một đỉnh.) Hơn nữa, mọi cạnh an toàn (u, v) của A sẽ nối các thành phần riêng biệt của G_A, bởi $A \cup \{(u, v)\}$ phải là phi chu trình.

Vòng lặp trong các dòng 2-4 của GENERIC-MST được thi hành $|V| - 1$ lần bởi từng trong số $|V| - 1$ cạnh của một cây tỏa nhánh cực tiểu liên tiếp được xác định. Thoạt đầu, khi $A = \emptyset$, ta có $|V|$ cây trong G_A, và mỗi lần lặp lại sẽ rút gọn số đó đi 1. Khi rừng chỉ chứa một cây đơn lẻ.

thuật toán kết thúc.

Hai thuật toán trong Đoạn 24.2 dùng hệ luận dưới đây cho Định lý 24.1.

Hệ luận 24.2

Cho $G = (V, E)$ là một đồ thị không hướng, liên thông có một hàm trọng số có giá trị thực w được định nghĩa trên E. Cho A là một tập hợp con của E nằm trong vài cây tỏa nhánh cực tiểu của G, và cho C là một thành phần liên thông (cây) trong rừng $G_A = (V, A)$. Nếu (u, v) là một cạnh nhạt nối C với một thành phần nào khác trong G_A, thì (u, v) an toàn đối với A.

Chứng minh Phần cắt $(C, V - C)$ tôn trọng A, và do đó (u, v) là một cạnh nhạt cho phần cắt này.

Bài tập

24.1-1

Cho (u, v) là một cạnh có trọng số cực tiểu trong một đồ thị G. Chứng tỏ (u, v) thuộc về một cây tỏa nhánh cực tiểu của G.

24.1-2

Giáo sư Sabatier giả định đảo đề sau đây của Định lý 24.1. Cho $G = (V, E)$ là một đồ thị không hướng, liên thông với một hàm trọng số có giá trị thực w được định nghĩa trên E. Cho A là một tập hợp con của E nằm trong một cây tỏa nhánh cực tiểu của G, cho $(S, V - S)$ là phần cắt bất kỳ của G tôn trọng A, và cho (u, v) là một cạnh an toàn để A đi qua $(S, V - S)$. Thì, (u, v) là một cạnh nhạt cho phần cắt. Chứng tỏ giả định của giáo sư không đúng bằng cách cho một ví dụ ngược lại.

24.1-3

Chứng tỏ nếu một cạnh (u, v) nằm trong một cây tỏa nhánh cực tiểu, thì nó là một cạnh nhạt đi qua một phần cắt của đồ thị.

24.1-4

Nêu một ví dụ đơn giản của một đồ thị sao cho tập hợp tất cả các cạnh là cạnh nhạt đi qua một phần cắt trong đồ thị sẽ không hình thành một cây tỏa nhánh cực tiểu.

24.1-5

Cho e là một cạnh trọng số cực đại trên một chu trình của $G = (V, E)$. Chứng minh có một cây tỏa nhánh cực tiểu của $G' = (V, E - \{e\})$ cũng là một cây tỏa nhánh cực tiểu của G.

24.1-6

Chứng tỏ một đồ thị có một cây tỏa nhánh cực tiểu duy nhất nếu, với mọi phần cắt của đồ thị, ta có một cạnh nhạt duy nhất đi qua phần cắt. Chứng tỏ đảo đề không đúng bằng cách đưa ra một ví dụ ngược lại.

24.1-7

Chứng tỏ nếu tất cả các trọng số cạnh của một đồ thị là dương, thì mọi tập hợp con các cạnh nối tất cả các đỉnh và có tổng trọng số cực tiểu phải là một cây. Nêu một ví dụ để chứng tỏ không thể theo cùng kết luận nếu ta cho phép vài trọng số không dương.

24.1-8

Cho T là một cây tỏa nhánh cực tiểu của một đồ thị G, và cho L là danh sách các trọng số cạnh được sắp xếp của T. Chứng tỏ với bất kỳ cây tỏa nhánh cực tiểu T' nào khác của G, danh sách L cũng là danh sách các trọng số cạnh sắp xếp của T'.

24.1-9

Cho T là một cây tỏa nhánh cực tiểu của một đồ thị $G = (V, E)$, và cho V' là một tập hợp con của V. Cho T' là đồ thị con của T được cảm sinh bởi V', và cho G' là đồ thị con của G được cảm sinh bởi V'. Chứng tỏ nếu T' được nối, thì T' là một cây tỏa nhánh cực tiểu của G'.

24.2 Thuật toán Kruskal và Prim

Hai thuật toán cây tỏa nhánh cực tiểu mô tả trong đoạn này là những dạng chi tiết của thuật toán chung. Chúng dùng một quy tắc cụ thể để xác định một cạnh an toàn trong dòng 3 của GENERIC-MST. Trong thuật toán Kruskal, tập hợp A là một rừng. Cạnh an toàn được bổ sung vào A luôn là một cạnh có trọng số ít nhất trong đồ thị nối hai thành phần riêng biệt. Trong thuật toán Prim, tập hợp A hình thành một cây đơn lẻ. Cạnh an toàn được bổ sung vào A luôn là một cạnh có trọng số ít nhất nối cây với một đỉnh không nằm trong cây.

Thuật toán Kruskal

Thuật toán Kruskal trực tiếp dựa trên thuật toán cây tỏa nhánh cực tiểu chung nêu trong Đoạn 24.1. Nó tìm một cạnh an toàn để bổ sung vào rừng đang tăng trưởng bằng cách tìm, trong số tất cả các cạnh nối hai cây bất kỳ trong rừng, một cạnh (u, v) có trọng số ít nhất. Cho C_1 và C_2 thể hiện hai cây được nối bởi (u, v). Do (u, v) phải là một cạnh nhạt

nối C_1 với một cây khác, nên Hệ luận 24.2 hàm ý (u, v) là một cạnh an toàn của C_1. Thuật toán Kruskal là một thuật toán tham, bởi tại mỗi bước nó bổ sung vào rừng một cạnh có trọng số khả dĩ ít nhất.

Kiểu thực thi thuật toán Kruskal của chúng ta giống như thuật toán để tính các thành phần liên thông từ Đoạn 22.1. Nó sử dụng một cấu trúc dữ liệu tập hợp rời để duy trì một vài tập hợp rời gồm các thành phần. Mỗi tập hợp chứa các đỉnh trong một cây của rừng hiện hành. Phép toán FIND-SET(u) trả về một thành phần đại diện từ tập hợp chứa u. Như vậy, ta có thể xác định hai đỉnh u *và* v có thuộc về cùng cây hay không bằng cách trắc nghiệm FIND-SET(u) có bằng FIND-SET(v) hay không. Thủ tục UNION sẽ hoàn thành tiến trình tổ hợp các cây.

```
MST-KRUSKAL(G, w)
1  A ← ∅
2  for mỗi đỉnh v ∈ V[G]
3        do MAKE-SET(v)
4  sắp xếp các cạnh của E theo trọng số không giảm w
5  for mỗi cạnh (u, v) ∈ E, theo thứ tự của trọng số không giảm
6        do if FIND-SET(u) ≠ FIND-SET(v)
7              then A ← A ∪ {(u, v)}
8                   UNION(u, v)
9  return A
```

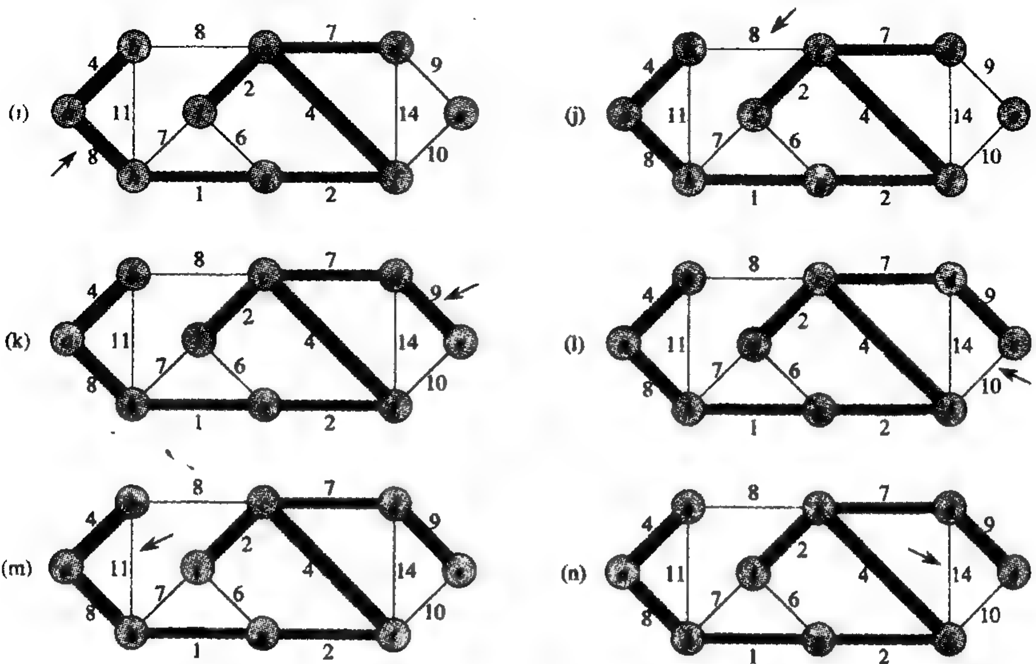
Thuật toán Kruskal làm việc như đã nêu trong Hình 24.4. Các dòng 1-3 khởi tạo tập hợp A theo tập hợp trống và tạo $|V|$ cây, mỗi cây chứa một đỉnh. Các cạnh trong E được sắp xếp theo thứ tự trọng số không giảm trong dòng 4. Vòng lặp **for** trong các dòng 5-8 sẽ kiểm tra xem, với mỗi cạnh (u, v), các điểm cuối u và v có thuộc về cùng cây hay không. Nếu có, cạnh (u, v) không thể bổ sung vào rừng mà không tạo một chu trình, và cạnh được loại bỏ. Bằng không, hai đỉnh thuộc về các cây khác nhau, cạnh (u, v) được bổ sung vào A trong dòng 7, và các đỉnh trong hai cây được trộn trong dòng 8.

Thời gian thực hiện của thuật toán Kruskal cho một đồ thị $G = (V, E)$ tùy thuộc vào cách thực thi của cấu trúc dữ liệu tập hợp rời. Ta sẽ mặc nhận cách thực thi rừng tập hợp rời của Đoạn 22.3 với heuristic nén lộ trình và hợp theo hạng, bởi nó là cách thực thi nhanh nhất theo tiệm cận đã biết. Tiến trình khởi tạo mất một thời gian $O(V)$, và thời gian để sắp xếp các cạnh trong dòng 4 là $O(E \lg E)$. Có $O(E)$ phép toán trên rừng tập hợp rời, mà tổng cộng mất $O(E\, \alpha(E, V))$ thời gian, ở đó α là nghịch đảo

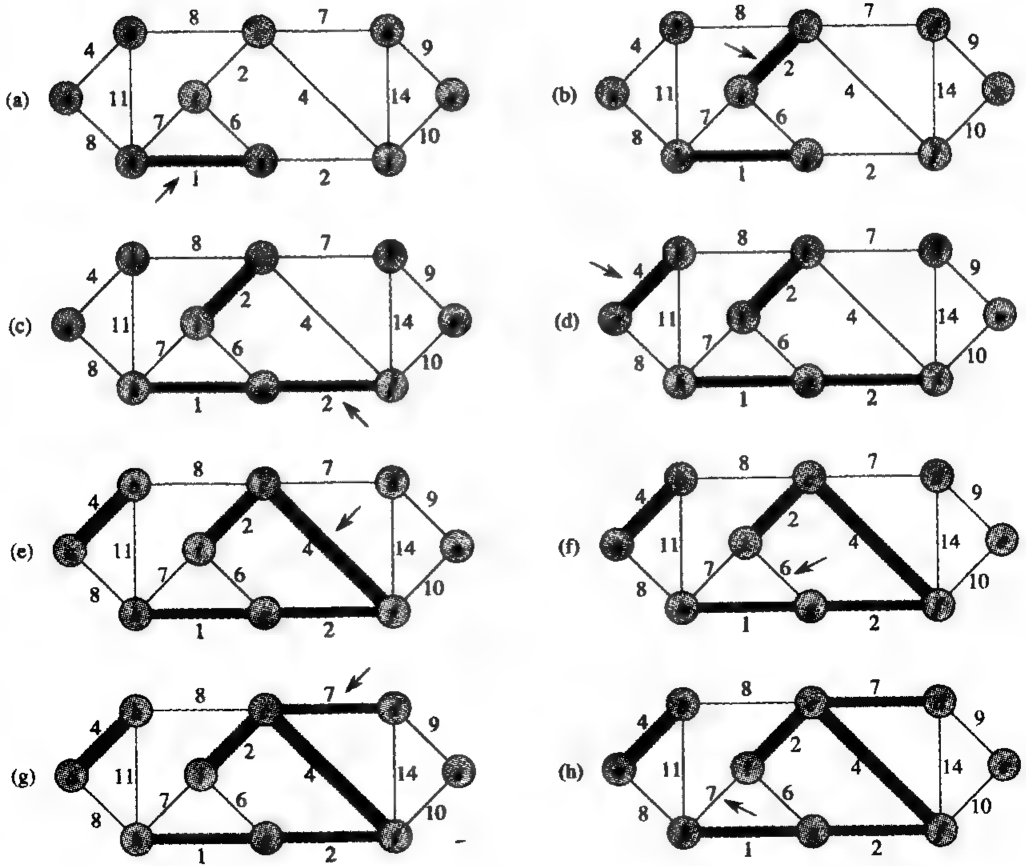
về chức năng với hàm Ackermann được định nghĩa trong Đoạn 22.4. Bởi $\alpha(E, V) = O(\lg E)$, tổng thời gian thực hiện của thuật toán Kruskal là $O(E \lg E)$.

Thuật toán Prim

Cũng như thuật toán Kruskal, thuật toán Prim là một trường hợp đặc biệt của thuật toán cây tỏa nhánh cực tiểu chung từ Đoạn 24.1. thuật toán Prim hoạt động tương tự như thuật toán Dijkstra để tìm các lộ trình ngắn nhất trong một đồ thị. (Xem Đoạn 25.2.) Thuật toán Prim có tính chất đó là các cạnh trong tập hợp A luôn hình thành một cây đơn lẻ. Như minh họa trong Hình 24.5, cây bắt đầu từ một đỉnh gốc tùy ý r và tăng trưởng cho đến khi cây tỏa rộng tất cả các đỉnh trong V. Tại mỗi bước, một cạnh nhạt nối một đỉnh trong A với một đỉnh trong $V - A$ sẽ được bổ sung vào cây. Theo Hệ luận 24.2, quy tắc này chỉ bổ sung các cạnh an toàn đối với A; do đó, khi thuật toán kết thúc, các cạnh trong A hình thành một cây tỏa nhánh cực tiểu. Chiến lược này là "tham" bởi cây được tăng cường tại mỗi bước bằng một cạnh đóng góp lượng cực tiểu khả dĩ vào trọng số của cây.



Hình 24.4 Thi hành thuật toán Kruskal trên đồ thị trong Hình 24.1. Các cạnh tô bóng thuộc về rừng A đang tăng trưởng. Các cạnh được thuật toán xét theo thứ tự sắp xếp trọng số. Một mũi tên trỏ đến cạnh đang xét tại mỗi bước của thuật toán. Nếu cạnh ghép hai cây riêng biệt trong rừng, nó được bổ sung vào rừng, và như vậy trộn hai cây.

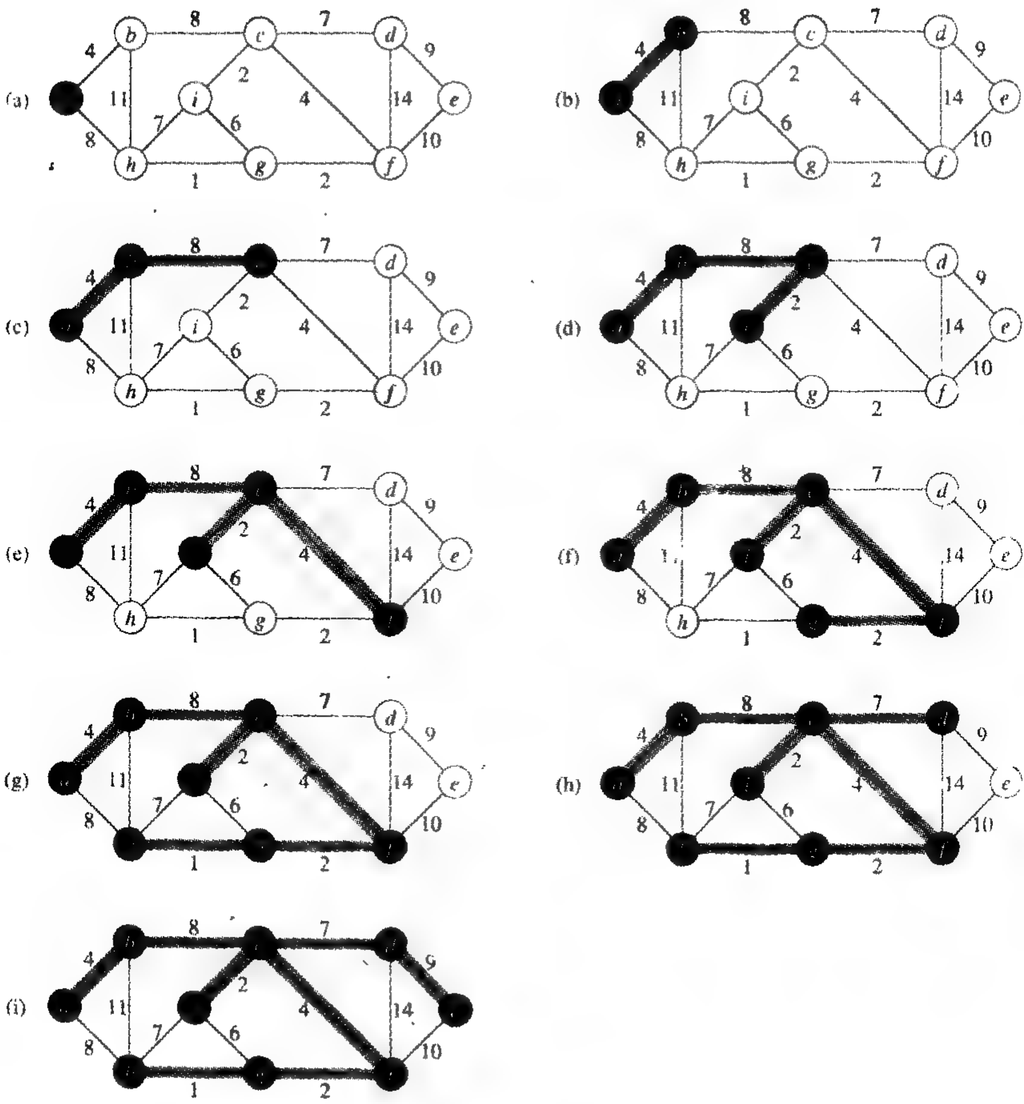


Chìa khóa để thực thi thuật toán Prim một cách hiệu quả đó là giúp ta dễ dàng lựa một cạnh mới để bổ sung vào cây mà các cạnh trong A hình thành. Trong mã giả dưới đây, đồ thị liên thông G và gốc r của cây tỏa nhánh cực tiểu được tăng trưởng sẽ làm nhập liệu cho thuật toán. Trong khi thi hành thuật toán, tất cả các đỉnh *không* nằm trong cây sẽ thường trú trong một hàng đợi ưu tiên Q dựa trên một trường *key*. Với mỗi đỉnh v, $key[v]$ là trọng số cực tiểu của bất kỳ cạnh nào nối v với một đỉnh trong cây; theo quy ước, $key[v] = \infty$ nếu không có cạnh nào như vậy. Trường $\pi[v]$ nêu tên "cha" của v trong cây. Trong thuật toán, tập hợp A của GENERIC-MST được duy trì mặc định dưới dạng

$$A = \{(v, \pi[v]) : v \in V - \{r\} - Q\} .$$

Khi thuật toán kết thúc, hàng đợi ưu tiên Q trống; như vậy, cây tỏa nhánh cực tiểu A của G là

$$A = \{(v, \pi[v]) : v \in V - \{r\}\} .$$



Hình 24.5 Thi hành thuật toán Prim trên đồ thị trong Hình 24.1. Đỉnh gốc là a. Các cạnh tô bóng nằm trong cây đang tăng trưởng, và các đỉnh trong cây được nêu ở dạng đen. Tại mỗi bước của thuật toán, các đỉnh trong cây xác định một phần cắt của đồ thị, và một cạnh nhạt đi qua phần cắt được bổ sung vào cây. Ví dụ, trong bước thứ hai, thuật toán có một chọn lựa bổ sung cạnh (b, c) hoặc cạnh (a, h) vào cây bởi cả hai là các cạnh nhạt đi qua phần cắt.

```
MST-PRIM(G, w, r)
1  Q ← V[G]
2  for mỗi u ∈ Q
3        do key[u] ← ∞
4  key[r] ← 0
5  π[r] ← NIL
6  while Q ≠ ∅
7        do u ← EXTRACT-MIN(Q)
8            for mỗi v ∈ Adj[u]
9                do if v ∈ Q và w(u, v) < key[v]
10                     then π[v] ← u
11                          key[v] ← w(u,v)
```

Thuật toán Prim làm việc như đã nêu trong Hình 24.5. Các dòng 1-4 khởi tạo hàng đợi ưu tiên Q để chứa tất cả các đỉnh và ấn định khóa của mỗi đỉnh theo ∞, ngoại trừ gốc r, có khóa được ấn định theo 0. Dòng 5 khởi tạo $\pi[r]$ theo NIL, bởi gốc r không có cha. Suốt thuật toán, tập hợp $V - Q$ chứa các đỉnh trong cây đang tăng trưởng. Dòng 7 định danh một đỉnh $u \in Q$ liên thuộc trên một cạnh nhạt đi qua phần cắt $(V - Q, Q)$ (với ngoại lệ là lần lặp lại đầu tiên, ở đó $u = r$ do dòng 4). Việc gỡ bỏ u ra khỏi tập hợp Q sẽ bổ sung nó vào tập hợp $V - Q$ gồm các đỉnh trong cây. Các dòng 8-11 cập nhật các trường *key* và π của mọi đỉnh v kề với u nhưng không nằm trong cây. Tiến trình cập nhật duy trì các bất biến $key[v] = w(v, \pi[v])$ và $(v, \pi[v])$ là một cạnh nhạt nối v với một đỉnh trong cây.

Khả năng thực hiện của thuật toán Prim tùy thuộc vào cách thực thi hàng đợi ưu tiên Q. Nếu Q được thực thi dưới dạng một đống nhị phân (xem Chương 7), ta có thể dùng thủ tục BUILD-HEAP để thực hiện tiến trình khởi tạo trong các dòng 14 trong $O(V)$ thời gian. Vòng lặp được thi hành $|V|$ lần, và do mỗi phép toán EXTRACT-MIN mất $O(\lg V)$ thời gian, nên tổng thời gian cho tất cả các lệnh gọi EXTRACT-MIN là $O(V \lg V)$. Vòng lặp **for** trong các dòng 8-11 được thi hành $O(E)$ lần, bởi tổng các chiều dài của tất cả các danh sách kề là $2|E|$. Trong vòng lặp **for**, có thể thực thi đợt trắc nghiệm tư cách thành viên trong Q trong dòng 9 trong thời gian bất biến bằng cách duy trì một bit cho mỗi đỉnh báo cho biết nó có nằm trong Q hay không, và cập nhật bit đó khi đỉnh được gỡ bỏ ra khỏi Q. Phép gán trong dòng 11 bao hàm một phép toán DECREASE-KEY mặc định trên đống, có thể được thực thi trong một

đống nhị phân trong $O(\lg V)$ thời gian. Như vậy, tổng thời gian của thuật toán Prim là $O(V \lg V + E \lg V) = O(E \lg V)$, mà theo tiệm cận giống như cách thực thi của thuật toán Kruskal.

Tuy nhiên, thời gian thực hiện tiệm cận của thuật toán Prim có thể được cải thiện bằng cách dùng các đống Fibonacci. Chương 21 chứng tỏ nếu $|V|$ thành phần được tổ chức thành một đống Fibonacci, ta có thể thực hiện một phép toán EXTRACT-MIN trong $O(\lg V)$ thời gian khấu trừ và một phép toán DECREASE-KEY (để thực thi dòng 11) trong $O(1)$ thời gian khấu trừ. Do đó, nếu ta dùng một đống Fibonacci để thực thi hàng đợi ưu tiên Q, thời gian thực hiện của thuật toán Prim sẽ được cải thiện thành $O(E + V \lg V)$.

Bài tập

24.2-1

Thuật toán Kruskal có thể trả về các cây tỏa nhánh khác nhau cho cùng đồ thị nhập liệu G, tùy thuộc vào cách thức mà các mối ràng buộc bị tách khi các cạnh được sắp xếp thành thứ tự. Chứng tỏ với mỗi cây tỏa nhánh cực tiểu T của G, ta có một cách để sắp xếp các cạnh của G trong thuật toán Kruskal sao cho thuật toán trả về T.

24.2-2

Giả sử rằng đồ thị $G = (V, E)$ được biểu thị dưới dạng một ma trận kề. Nêu một cách thực thi đơn giản của thuật toán Prim cho trường hợp này chạy trong $O(V^2)$ thời gian.

24.2-3

Theo tiệm cận, cách thực thi đống Fibonacci của thuật toán Prim có nhanh hơn cách thực thi đống nhị phân của một đồ thị thưa $G = (V, E)$, ở đó $|E| = \Theta(V)$ hay không? Với một đồ thị trù mật, ở đó $|E| = \Theta(V^2)$, thì sao? $|E|$ và $|V|$ phải quan hệ như thế nào để theo tiệm cận cách thực thi đống Fibonacci nhanh hơn cách thực thi đống nhị phân?

24.2-4

Giả sử rằng tất cả các trọng số cạnh trong một đồ thị là các số nguyên trong miền giá trị từ 1 đến $|V|$. Bạn có thể khiến thuật toán Kruskal chạy nhanh đến mức nào? Điều gì xảy ra nếu các trọng số cạnh là các số nguyên trong miền giá trị từ 1 đến W với một hằng W nào đó?

24.2-5

Giả sử rằng tất cả các trọng số cạnh trong một đồ thị là các số nguyên trong miền giá trị từ 1 các $|V|$. Bạn có thể tạo thuật toán Prim chạy nhanh

đến mức nào? Điều gì xảy ra nếu các trọng số cạnh là các số nguyên trong miền giá trị từ 1 đến W với một hằng W nào đó?

24.2-6

Mô tả một thuật toán hiệu quả mà, căn cứ vào một đồ thị không hướng G, xác định một cây tỏa nhánh của G mà trọng số cạnh lớn nhất của nó là cực tiểu trên tất cả các cây tỏa nhánh của G.

24.2-7*

Giả sử rằng các trọng số cạnh trong một đồ thị được phân phối đồng đều trên quãng nửa mở [0, 1). Bạn có thể làm cho thuật toán nào chạy nhanh hơn, Kruskal hay Prim?

24.2-8*

Giả sử rằng một đồ thị G có một cây tỏa nhánh cực tiểu đã tính toán sẵn. Có thể cập nhật cây tỏa nhánh cực tiểu nhanh tới mức nào nếu một đỉnh mới và các cạnh liên thuộc được bổ sung vào G?

Các Bài toán

24-1 Cây tỏa nhánh cực tiểu tốt nhất thứ hai

Cho $G = (V, E)$ là một đồ thị liên thông không hướng có hàm trọng số $w : E \rightarrow \mathbf{R}$, và giả sử rằng $|E| \geq |V|$.

a. Cho T là một cây tỏa nhánh cực tiểu của G. Chứng minh ở đó tồn tại các cạnh $(u, v) \in T$ và $(x, y) \notin T$ sao cho $T - \{(u, v)\} \cup \{(x, y)\}$ là một cây tỏa nhánh cực tiểu tốt nhất thứ hai của G.

b. Cho T là một cây tỏa nhánh của G và, với bất kỳ hai đỉnh $u, v \in V$, cho $max[u, v]$ là một cạnh có trọng số cực đại trên lộ trình duy nhất giữa u và v trong T. Mô tả một thuật toán $O(V^2)$ thời gian mà, căn cứ vào T, tính toán $max[u, v]$ với tất cả $u, v \in V$.

c. Nêu một thuật toán hiệu quả để tính cây tỏa nhánh cực tiểu tốt nhất thứ hai của G.

24-2 Cây tỏa nhánh cực tiểu trong các đồ thị thưa

Với một đồ thị liên thông rất thưa $G = (V, E)$, ta có thể cải thiện thời gian thực hiện $O(E + V \lg V)$ của thuật toán Prim có các đống Fibonacci bằng cách "xử lý trước" G để giảm số lượng các đỉnh trước khi chạy thuật toán Prim. Thủ tục dưới đây chấp nhận một đồ thị gia trọng G làm nhập liệu và trả về một phiên bản "rút gọn" của G, đã bổ sung vài cạnh

vào cây tỏa nhánh cực tiểu T đang kiến tạo. Thoạt đầu, với mỗi cạnh $(u, v) \in E$, ta mặc nhận $orig[u, v] = (u, v)$ và $w[u, v]$ là trọng số của cạnh.

```
MST-REDUCE(G, T)
for mỗi v ∈ V[G]
     do mark[v] ← FALSE
        MAKE-SET(v)
for mỗi u ∈ V[G]
     do if mark[u] = FALSE
           then chọn v ∈ Adj[u] sao cho w[u, v] được cực tiểu hóa
                UNION(u, v)
                T ← T ∪ {orig[u, v]}
                mark[u] ← mark[v] ← TRUE
V[G'] ← {FIND-SET(v) : v ∈ V[G]}
E[G'] ← ∅
for mỗi (x, y) ∈ E[G]
     do u ← FIND-SET(x)
        v ← FIND-SET(y)
        if (u, v) ∉ E[G]
          then E[G'] ← E[G'] ∪ {(u, v)}
               orig[u, v] ← orig[x, y]
               w[u,v] ← w[x,y]
          else if w[x, y] < w[u, v]
                  then orig[u, v] ← orig[x, y]
                       w[u, v] ← w[x, y]
kiến tạo các danh sách kề Adj cho G'
return G' và T
```

a. Cho T là tập hợp các cạnh được MST-REDUCE trả về, và cho T' là một cây tỏa nhánh cực tiểu của đồ thị G' do thủ tục trả về. Chứng minh $T \cup \{orig[x, y] : (x, y) \in T'\}$ là một cây tỏa nhánh cực tiểu của G.

b. Chứng tỏ $|V[G']| < |V|/2$.

c. Nêu cách thực thi MST-REDUCE sao cho nó chạy trong $O(E)$ thời gian. (*Mách nước:* Dùng các cấu trúc dữ liệu đơn giản.)

d. Giả sử ta chạy k giai đoạn của MST-REDUCE, dùng đồ thị do một

giai đoạn tạo ra làm nhập liệu cho giai đoạn kế tiếp và tích lũy các cạnh trong T. Chứng tỏ thời gian thực hiện chung của k giai đoạn là $O(kE)$.

e. Giả sử rằng sau khi chạy k giai đoạn của MST-REDUCE, ta chạy thuật toán Prim trên đồ thị mà giai đoạn chót trả về. Nêu cách chọn k sao cho thời gian thực hiện chung là $O(E \lg \lg V)$. Chứng tỏ sự chọn lựa của bạn về k giảm thiểu thời gian thực hiện tiệm cận chung.

f. Nêu các giá trị của $|E|$ (theo dạng $|V|$) mà thuật toán Prim có tiến trình tiền xử lý theo tiệm cận sẽ đánh bại thuật toán Prim không có tiến trình tiền xử lý?

Ghi chú Chương

Tarjan [188] nghiên cứu bài toán cây tỏa nhánh cực tiểu và cung cấp các chất liệu cao cấp tuyệt vời. Graham và Hell [92] có ghi lại một lịch sử về bài toán cây tỏa nhánh cực tiểu.

Tarjan quy thuật toán cây tỏa nhánh cực tiểu đầu tiên cho một tài liệu 1926 của O. Boruvka. Thuật toán Kruskal được Kruskal [131] báo cáo vào năm 1956. Thuật toán thường được xem là thuật toán Prim đã được Prim [163] sáng chế, nhưng nó cũng được sáng chế trước đó bởi V. Jarnik Vào năm 1930.

Lý do tại sao các thuật toán tham lại hiệu quả đối với việc tìm các cây tỏa nhánh cực tiểu đó là vì tập hợp các rừng của một đồ thị đã hình thành một tạng ma trận đồ họa. (Xem Đoạn 17.4.)

Thuật toán cây tỏa nhánh cực tiểu nhanh nhất hiện nay dành cho trường hợp ở đó $|E| = \Omega(V \lg V)$ đó là thuật toán Prim được thực thi với các đống Fibonacci. Với đồ thị thưa hơn, Fredman và Tarjan [75] cho ra một thuật toán chạy trong $O(E\,\beta(|E|, |V|))$ thời gian, ở đó $\beta(|E|, |V|) = \min\{i : \lg^{(i)} |V| \leq |E|/|V|\}$. Việc $|E| \geq |V|$ hàm ý thuật toán của họ chạy trong thời gian $O(E \lg^* V)$.

25 Các Lộ Trình Ngắn Nhất Nguồn Đơn

Một tài xế ô tô muốn tìm tuyến đường khả dĩ ngắn nhất từ Chicago đến Boston. Cho một bản đồ đường xá của nước Mỹ qua đó khoảng cách giữa mỗi cặp giao lộ kề nhau được đánh dấu, làm sao để có thể xác định tuyến đường ngắn nhất này?

Một cách khả dĩ đó là điểm danh tất cả các tuyến đường từ Chicago đến Boston, cộng dồn các khoảng cách trên mỗi tuyến đường, và lựa tuyến ngắn nhất. Tuy nhiên, sẽ dễ dàng nhận ra rằng cho dù không cho phép các tuyến đường chứa các chu trình, ta vẫn có hàng triệu khả năng, hầu hết chúng chẳng đáng để xem xét. Ví dụ, một tuyến đường từ Chicago đến Houston đến Boston hiển nhiên là một chọn lựa tồi, bởi Houston dôi ra khoảng một ngàn dặm.

Trong chương này và Chương 26, ta nêu cách giải quyết các bài toán này một cách hiệu quả. Trong một ***bài toán các lộ trình ngắn nhất***, ta có một đồ thị có hướng gia trọng $G = (V, E)$, với hàm trọng số $w : E \rightarrow \mathbf{R}$ ánh xạ các cạnh theo các trọng số có giá trị thực. ***Trọng số*** của lộ trình $p = \langle v_0, v_1, \ldots, v_k \rangle$ là tổng của các trọng số của các cạnh cấu thành của nó:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) .$$

Ta định nghĩa ***trọng số lộ trình ngắn nhất*** [shortest-path weight] từ u đến v theo

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\leadsto} v\} & \text{nếu có một lộ trình từ } u \text{ đến } v , \\ \infty & \text{bằng không} . \end{cases}$$

Như vậy, một ***lộ trình ngắn nhất*** từ đỉnh u đến đỉnh v được định nghĩa là bất kỳ lộ trình p nào có trọng số $w(p) = \delta(u,v)$.

Trong ví dụ Chicago đến Boston, ta có thể lập mô hình bản đồ đường xá dưới dạng một đồ thị: các đỉnh biểu diễn các giao lộ, các cạnh biểu diễn các đoạn đường giữa các giao lộ, và các trọng số cạnh biểu diễn các khoảng cách đường. Mục tiêu của chúng ta đó là tìm một lộ trình ngắn nhất từ một giao lộ đã cho trong Chicago (giả sử, Clark St. và

Addison Ave.) đến một giao lộ đã cho trong Boston (giả sử, Brookline Ave. và Yawkey Way).

Có thể diễn dịch các trọng số cạnh dưới dạng metric thay vì các khoảng cách. Chúng thường được dùng để biểu diễn thời gian, mức hao phí, các hình phạt, sự mất mát, hoặc bất kỳ số lượng nào khác tích lũy theo tuyến tính dọc theo một lộ trình và ta muốn giảm thiểu.

Thuật toán tìm độ rộng đầu tiên trong Đoạn 23.2 là một thuật toán các lộ trình ngắn nhất làm việc trên các đồ thị không gia trọng, nghĩa là, các đồ thị ở đó mỗi cạnh có thể được xem là có trọng số đơn vị. Do có nhiều khái niệm từ thuật toán tìm kiếm độ rộng đầu tiên nảy sinh trong khi nghiên cứu các lộ trình ngắn nhất trong đồ thị gia trọng, nên độc giả đừng quên xem lại Đoạn 23.2 trước khi tiếp tục.

Các biến thức

Trong chương này, ta sẽ tập trung vào ***bài toán các lộ trình ngắn nhất nguồn đơn***:

căn cứ vào một đồ thị $G = (V, E)$, ta muốn tìm một lộ trình ngắn nhất từ một đỉnh ***nguồn*** $s \in V$ đã cho đến mọi đỉnh $v \in V$. Có thể giải quyết nhiều bài toán khác bằng thuật toán của bài toán nguồn đơn, kể cả các biến thức dưới đây.

Bài toán các lộ trình ngắn nhất đích đơn: Tìm một lộ trình ngắn nhất đến một đỉnh ***đích*** t từ mọi đỉnh v. Bằng cách đảo ngược hướng của mỗi cạnh trong đồ thị, ta có thể rút gọn bài toán này thành một bài toán nguồnđơn.

Bài toán lộ trình ngắn nhất cặp đơn: Tìm một lộ trình ngắn nhất từ u đến v với các đỉnh đã cho u và v. Nếu ta giải quyết bài toán nguồn đơn với đỉnh nguồn u, ta cũng giải quyết bài toán này. Hơn nữa, ta chưa từng biết có một thuật toán nào cho bài toán này mà theo tiệm cận chạy nhanh hơn các thuật toán nguồn đơn tốt nhất trong ca xấu nhất.

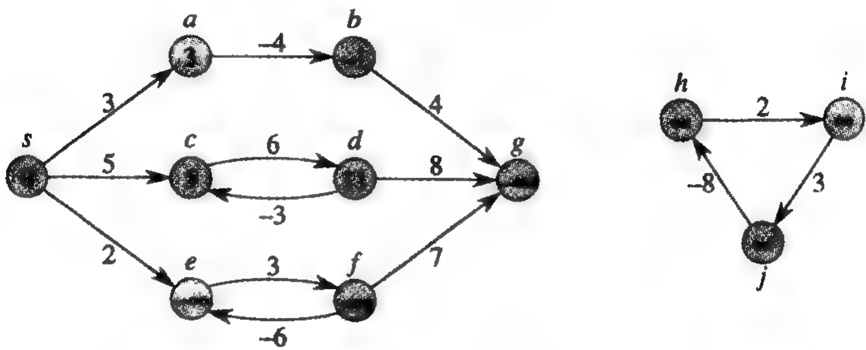
Bài toán các lộ trình ngắn nhất mọi cặp: Tìm một lộ trình ngắn nhất từ u đến v cho mọi cặp các đỉnh u và v. Có thể giải quyết bài toán này bằng cách chạy một thuật toán nguồnđơn một lần từ mỗi đỉnh; nhưng thông thường ta có thể giải quyết nó nhanh hơn, và cấu trúc củanó tự thân cũng đáng quan tâm. Chương 26 đề cập chi tiết bài toán mọi cặp.

Các cạnh trọng số âm

Trong vài trường hợp của bài toán các lộ trình ngắn nhất nguồnđơn, ta có thể có các cạnh mà các trọng số của chúng là âm. Nếu đồ thị $G = (V, E)$ không chứa các chu trình trọng số âm khả dụng từ nguồn s, thì với tất cả $v \in V$, trọng số lộ trình ngắn nhất $\delta(s, v)$ vẫn được định nghĩa tốt.

cho dù nó có một giá trị âm. Tuy nhiên, nếu có một chu trình trọng số âm khả dụng từ *s*, các trọng số lộ trình ngắn nhất không được định nghĩa tốt. Không có lộ trình nào từ *s* đến một đỉnh trên chu trình có thể là lộ trình ngắn nhất—luôn có thể tìm thấy một lộ trình có trọng số ít hơn theo sau lộ trình "ngắn nhất" đề xuất rồi đi qua chu trình có trọng số âm. Nếu có một chu trình trọng số âm trên một lộ trình từ *s* đến *v*, ta định nghĩa $\delta(s, v) = -\infty$.

Hình 25.1 minh họa hiệu ứng của các trọng số âm trên các trọng số lộ trình ngắn nhất. Bởi chỉ có một lộ trình từ *s* đến *a* (lộ trình <*s*, *a*>), $\delta(s, a) = w(s, a) = 3$. Cũng vậy, chỉ có một lộ trình từ *s* đến *b*, và do đó $\delta(s, b) = w(s, a) + w(a, b) = 3 + (-4) = -1$. Có rất nhiều lộ trình từ *s* đến *c*: <*s*, *c*>, <*s*, *c*, *d*, *c*>, <*s*, *c*, *d*, *c*, *d*, *c*>, và vân vân. Bởi chu trình <*c*, *d*, *c*> với trọng số $6 + (-3) = 3 > 0$, nên lộ trình ngắn nhất từ *s* đến *c* là (*s*, *c*), với trọng số $\delta(s, c) = 5$. Cũng vậy, lộ trình ngắn nhất từ *s* đến *d* là <*s*, *c*, *d*>, với trọng số $\delta(s, d) = w(s, c) + w(c, d) = 11$. Tương tự, có rất nhiều lộ trình từ *s* đến *e*: <*s*, *e*>, <*s*, *e*, *f*, *e*>, <*s*, *e*, *f*, *e*, *f*, *e*>, và vân vân. Tuy nhiên, do chu trình (*e*, *f*, *e*) có trọng số $3 + (-6) = -3 < 0$, ta không có lộ trình ngắn nhất từ *s* đến *e*. Nhờ đi qua chu trình trọng số âm <*e*, *f*, *e*> nhiều lần một cách tùy ý, ta có thể tìm thấy các lộ trình từ *s* đến *e* có các trọng số âm lớn tùy ý, và do đó $\delta(s, e) = -\infty$. Cũng vậy, $\delta(s, f) = -\infty$. Bởi *g* khả dụng từ *f*, nên ta cũng có thể tìm các lộ trình có các trọng số âm lớn tùy ý từ *s* đến *g*, và $\delta(s, g) = -\infty$. Các đỉnh *h*, *i*, và *j* cũng hình thành một chu trình trọng số âm. Tuy nhiên, chúng không khả dụng từ *s*, và do đó $\delta(s, h) = \delta(s, i) = \delta(s, j) = \infty$.



Hình 25.1 Các trọng số cạnh âm trong một đồ thị có hướng. Được nêu trong mỗi đỉnh là trọng số lộ trình ngắn nhất của nó từ các nguồn *s*. Do các đỉnh *e* và *f* hình thành một chu trình trọng số âm khả dụng từ *s*, chúng có các trọng số lộ trình ngắn nhất của $-\infty$. Do đỉnh *g* khả dụng từ một đỉnh có trọng số lộ trình ngắn nhất là $-\infty$, nên nó cũng có một trọng số lộ trình ngắn nhất của $-\infty$. Các đỉnh như *h*, *i*, và *j* là không khả dụng từ *s*, và do đó các trọng số lộ trình ngắn nhất của chúng là ∞, cho dù chúng nằm trên một chu trình trọng số âm.

Vài thuật toán lộ trình ngắn nhất, như thuật toán Dijkstra, mặc nhận rằng tất cả các trọng số cạnh trong đồ thị nhập liệu là không âm, như trong ví dụ bản đồ đường. Các thuật toán khác, như thuật toán Bellman-Ford, cho phép các cạnh trọng số âm trong đồ thị nhập liệu và tạo một đáp án đúng đắn miễn là không có các chu trình trọng số âm khả dụng từ nguồn. Thông thường, nếu có một chu trình trọng số âm như vậy, thuật toán có thể phát hiện và báo cáo sự hiện diện của nó.

Biểu thị các lộ trình ngắn nhất

Thông thường ta muốn tính toán không những các trọng số lộ trình ngắn nhất, mà còn các đỉnh trên các lộ trình ngắn nhất. Phép biểu diễn mà ta dùng cho các lộ trình ngắn nhất cũng tương tự như trường hợp các cây độ rộng đầu tiên trong Đoạn 23.2. Cho một đồ thị $G = (V, E)$, ta duy trì với mỗi đỉnh $v \in V$ một ***phần tử tiền vị*** $\pi[v]$ là một đỉnh khác hoặc NIL. Các thuật toán lộ trình ngắn nhất trong chương này ấn định các thuộc tính π sao cho xích của các phần tử tiền vị phát sinh tại một đỉnh v chạy lùi dọc theo một lộ trình ngắn nhất từ s đến v. Như vậy, cho một đỉnh v mà $\pi[v] \neq$ NIL, thủ tục PRINT-PATH (G,s,v) trong Đoạn 23.2 có thể dùng để in một lộ trình ngắn nhất từ s đến v.

Tuy nhiên, trong khi thi hành một thuật toán các lộ trình ngắn nhất, các giá trị π không cần nêu rõ các lộ trình ngắn nhất. Như trong thuật toán tìm kiếm độ rộng đầu tiên, ta sẽ quan tâm ***đồ thị con phần tử tiền vị*** $G_\pi = (V_\pi, E_\pi)$ được cảm sinh bởi các giá trị π. Ở đây cũng vậy, ta định nghĩa tập hợp đỉnh V_π là tập hợp các đỉnh của G với các phần tử tiền vị phi-NIL, cộng với nguồn s:

$$V_\pi = \{v \in V: \pi[v] \neq \text{NIL}\} \cup \{s\} .$$

Tập hợp cạnh có hướng ấn định E_π là tập hợp các cạnh được cảm sinh bởi các giá trị π với các đỉnh trong V_π:

$$E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\} .$$

Ta sẽ chứng minh các giá trị π mà các thuật toán tạo trong chương này đều có tính chất rằng vào lúc kết thúc G_π là một "cây các lộ trình ngắn nhất "—không chính thức, một cây có gốc chứa một lộ trình ngắn nhất từ một nguồn s đến mọi đỉnh khả dụng từ s. Một cây các lộ trình ngắn nhất cũng giống như cây độ rộng đầu tiên trong Đoạn 23.2, nhưng nó chứa các lộ trình ngắn nhất từ nguồn được định nghĩa theo dạng các trọng số cạnh thay vì các số cạnh. Để chính xác, ta cho $G = (V, E)$ là một đồ thị có hướng gia trọng có hàm trọng số $w : E \to \mathbf{R}$, và mặc nhận rằng G không chứa các chu trình trọng số âm nào khả dụng từ đỉnh nguồn $s \in V$, sao cho các lộ trình ngắn nhất được định nghĩa tốt. Một ***cây các lộ trình ngắn nhất*** có gốc tại s là một đồ thị con có hướng $G' = (V', E')$, ở

đó $V' \subseteq V$ và $E' \subseteq E$, sao cho

1. V' là tập hợp các đỉnh khả dụng từ s trong G,

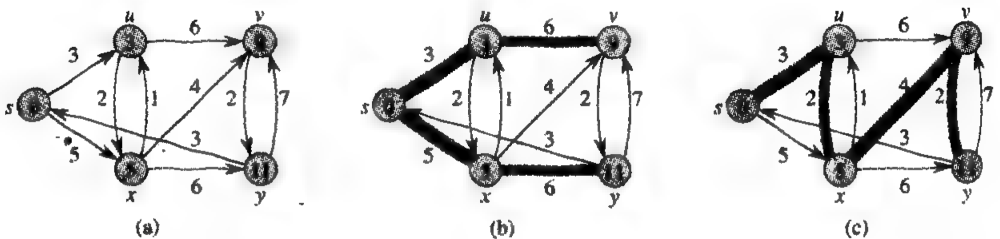
2. G' hình thành một cây có gốc với gốc s, và

3. với tất cả $v \in V'$, lộ trình đơn giản duy nhất từ s đến v trong G' là một lộ trình ngắn nhất từ s đến v trong G.

Các lộ trình ngắn nhất không nhất thiết là duy nhất, và cũng không phải là các cây các lộ trình ngắn nhất. Ví dụ, Hình 25.2 có nêu một đồ thị có hướng gia trọng và hai cây các lộ trình ngắn nhất có cùng gốc.

Khái quát chương

Tất cả các thuật toán lộ trình ngắn nhất nguồnđơn trong chương này đều dựa trên một kỹ thuật có tên là phép nới lỏng [relaxation]. Đoạn 25.1 bắt đầu bằng phần chứng minh vài tính chất quan trọng của các lộ trình ngắn nhất nói chung rồi chứng minh vài sự việc quan trọng về các thuật toán gốc nới lỏng. Đoạn 25.2 trình bày thuật toán Dijkstra, giải quyết bài toán các lộ trình ngắn nhất nguồn đơn khi tất cả các cạnh có trọng số không âm. Đoạn 25.3 trình bày thuật toán Bellman-Ford, được dùng trong trường hợp chung hơn ở đó các cạnh có thể có trọng số âm. Nếu đồ thị chứa một chu trình trọng số âm khả dụng từ nguồn, thuật toán Bellman-Ford phát hiện sự hiện diện củanó. Đoạn 25.4 cung cấp một thuật toán thời gian tuyến tính để tính các lộ trình ngắn nhất từ một nguồn đơn lẻ trong đồ thị phi chu trình có hướng. Cuối cùng, Đoạn 25.5 nêu cách dùng thuật toán Bellman-Ford để giải quyết một trường hợp đặc biệt của "lập trình tuyến tính."



Hình 25.2 (a) Một đồ thị có hướng gia trọng có các trọng số lộ trình ngắn nhất từ nguồn s. **(b)** Các cạnh tô bóng hình thành một cây các lộ trình ngắn nhất có gốc tại nguồn s. **(c)** Một cây các lộ trình ngắn nhất khác có cùng gốc.

Cuộc phân tích của chúng ta yêu cầu vài quy ước để thực hiện số học với các vô hạn. Ta sẽ mặc nhận rằng với bất kỳ số thực $a \neq -\infty$, ta có $a + \infty = \infty + a = \infty$. Ngoài ra, để các chứng minh của chúng ta đứng vững trong sự hiện diện của các chu trình trọng số âm, ta mặc nhận rằng

với bất kỳ số thực $a \neq \infty$, ta có $a + (-\infty) = (-\infty) + a = -\infty$.

25.1 Các lộ trình ngắn nhất và phép nới lỏng

Để hiểu rõ các thuật toán lộ trình ngắn nhất nguồn đơn, ta nên hiểu rõ các kỹ thuật mà chúng dùng và các tính chất của các lộ trình ngắn nhất mà chúng khai thác. Kỹ thuật chính được các thuật toán dùng trong chương này đó là phép nới lỏng, một phương pháp giảm liên tục một cận trên trên trọng số lộ trình ngắn nhất thực tế của mỗi đỉnh cho đến khi cận trên bằng trọng số lộ trình ngắn nhất. Trong đoạn này, ta sẽ xem cách làm việc của phép nới lỏng và chính thức chứng minh vài tính chất mà nó duy trì.

Với lần đọc đầu tiên của đoạn này, bạn có thể muốn bỏ qua các chứng minh của các định lý—chỉ đọc các câu lệnh của chúng—rồi tiếp tục đọc ngay các thuật toán trong các Đoạn 25.2 và 25.3. Tuy nhiên, bạn nên đặc biệt chú ý đến Bổ đề 25.7, là một chìa khóa để hiểu rõ các thuật toán các lộ trình ngắn nhất trong chương này. Trong lần đọc đầu tiên, có thể bạn cũng muốn bỏ qua hẳn các bổ đề liên quan đến các đồ thị con phần tử tiền vị và các cây các lộ trình ngắn nhất (Các bổ đề 25.8 và 25.9), thay vì thế tập trung vào các bổ đề trên đây, gắn liền với các trọng số lộ trình ngắn nhất.

Cấu trúc con tối ưu của một lộ trình ngắn nhất

Các thuật toán các lộ trình ngắn nhất thường khai thác tính chất mà một lộ trình ngắn nhất giữa hai đỉnh chứa các lộ trình ngắn nhất khác trong nó. Tính chất cấu trúc con tối ưu này là một dấu chất lượng về khả năng áp dụng của lập trình động (Chương 16) lẫn phương pháp tham (Chương 17). Thực vậy, thuật toán Dijkstra là một thuật toán tham, và thuật toán Floyd-Warshall, tìm các lộ trình ngắn nhất giữa tất cả cặp đỉnh (xem Chương 26), là một thuật toán lập trình động. Bổ đề dưới đây và hệ luận củanó phát biểu tính chất cấu trúc con tối ưu của các lộ trình ngắn nhất một cách chính xác hơn.

Bổ đề 25.1 (Các lộ trình con của các lộ trình ngắn nhất là các lộ trình ngắn nhất)

Cho một đồ thị có hướng gia trọng $G = (V, E)$ có hàm trọng số $w: E \rightarrow \mathbf{R}$, cho $p = \langle v_1, v_2, \ldots, v_k \rangle$ là một lộ trình ngắn nhất từ đỉnh v_1 đến đỉnh v_k và, với bất kỳ i và j sao cho $1 \leq i \leq j \leq k$, cho $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ là lộ trình con của p từ đỉnh v_i đến đỉnh v_j. Thì, p_{ij} là một lộ trình ngắn nhất từ v_i đến v_j.

Chứng minh Nếu ta phân tích lộ trình p thành $v_1 \overset{p_{1i}}{\leadsto} v_i \overset{p_{ij}}{\leadsto} v_j \overset{p_{jk}}{\leadsto} v_k$, thì $w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$. Giờ đây, mặc nhận rằng có một lộ trình p'_{ij} từ v_i đến v_j có trọng số $w(p'_{ij}) < w(p_{ij})$. Thì, $v_1 \overset{p_{1i}}{\leadsto} v_i \overset{p'_{ij}}{\leadsto} v_j \overset{p_{jk}}{\leadsto} v_k$ là một lộ trình từ v_1 đến v_k có trọng số $w(p_{1i}) + w(p'_{ij}) + w(p_{jk})$ nhỏ hơn $w(p)$, mâu thuẫn với tiền đề cho rằng p là một lộ trình ngắn nhất từ v_1 đến v_k.

Trong khi nghiên cứu thuật toán tìm kiếm độ rộng đầu tiên (Đoạn 23.2), ta đã chứng minh dưới dạng Bổ đề 23.1 một tính chất đơn giản của các khoảng cách ngắn nhất trong các đồ thị không gia trọng. Hệ luận dưới đây cho Bổ đề 25.1 tổng quát hóa tính chất cho các đồ thị gia trọng.

Hệ luận 25.2

Cho $G = (V, E)$ là một đồ thị có hướng gia trọng với hàm trọng số $w: E \rightarrow \mathbf{R}$. Giả sử rằng một lộ trình ngắn nhất p từ một nguồn s đến một đỉnh v có thể được phân tích thành $s \overset{p'}{\leadsto} u \rightarrow v$ với một đỉnh u và lộ trình p'. Thì, trọng số của một lộ trình ngắn nhất từ s đến v là $\delta(s, v) = \delta(s, u) + w(u, v)$.

Chứng minh Theo Bổ đề 25.1, lộ trình con p' là một lộ trình ngắn nhất từ nguồn s đến đỉnh u. Như vậy,

$$\begin{aligned} \delta(s,v) &= w(p) \\ &= w(p') + w(u,v) \\ &= \delta(s,u) + w(u,v) . \end{aligned}$$

Bổ đề kế tiếp đưa ra một tính chất tuy đơn giản nhưng hữu ích của các trọng số lộ trình ngắn nhất.

Bổ đề 25.3

Cho $G = (V, E)$ là một đồ thị có hướng gia trọng $G = (V, E)$ với hàm trọng số $w : E \rightarrow \mathbf{R}$ và đỉnh nguồn s. Thì, với tất cả các cạnh $(u, v) \in E$, ta có $\delta(s,v) \leq \delta(s,u) + w(u,v)$.

Chứng minh Một lộ trình ngắn nhất p từ nguồn s đến đỉnh v không có nhiều trọng số hơn bất kỳ lộ trình nào khác từ s đến v. Cụ thể, lộ trình p không có nhiều trọng số hơn lộ trình cụ thể chấp nhận một lộ trình ngắn nhất từ nguồn s đến đỉnh u rồi chấp nhận cạnh (u, v).

Phép nới lỏng

Các thuật toán trong chương này dùng kỹ thuật của ***phép nới lỏng*** [relaxation]. Với mỗi đỉnh $v \in V$, ta duy trì một thuộc tính $d[v]$, là một cận trên trên trọng số của một lộ trình ngắn nhất từ nguồn s đến v. Ta gọi $d[v]$ là ***ước lượng lộ trình ngắn nhất***. Ta khởi tạo các phần tử tiền vị

và các ước lượng lộ trình ngắn nhất và bằng thủ tục dưới đây.

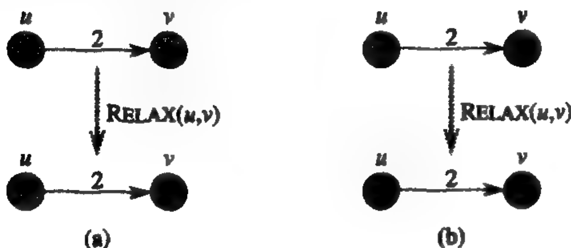
```
INITIALIZE-SINGLE-SOURCE(G, s)
for mỗi đỉnh v ∈ V[G]
do d[v] ← ∞
    π[v] ← NIL
d[s] ← 0
```

Sau khi khởi tạo, $\pi[v]$ = NIL với mọi $v \in V$, $d[v] = 0$ với $v = s$, và $d[v] = \infty$ với $v \in V - \{s\}$.

Tiến trình ***nới lỏng***[1] một cạnh (u, v) bao gồm việc kiểm tra xem ta có thể cải thiện lộ trình ngắn nhất đến v tìm thấy cho đến giờ bằng cách đi qua u hay không và, nếu có, cập nhật $d[v]$ và $\pi[v]$. Một bước nới lỏng có thể giảm giá trị của mức ước lượng lộ trình ngắn nhất $d[v]$ và cập nhật trường phần tử tiền vị của v $\pi[v]$. Mã dưới đây thực hiện một bước nới lỏng trên cạnh (u, v).

```
RELAX (u, v, w)
if d[v] > d[u] + w(u, v)
    then d[v] ← d[u] + w(u,v)
         π[v] ← u
```

Hình 25.3 nêu hai ví dụ nới lỏng một cạnh, một ví dụ ở đó một mức ước lượng lộ trình ngắn nhất giảm và một ví dụ ở đó không có ước lượng nào thay đổi.



Hình 25.3 Phép nới lỏng của một cạnh (u, v). Ước lượng lộ trình ngắn nhất của mỗi đỉnh được nêu trong đỉnh. **(a)** Bởi $d[v] > d[u] + w(u,v)$ trước phép nới lỏng, nên giá trị của $d[v]$ giảm. **(b)** Ở đây, $d[v] \le d[u] + w(u, v)$ trước bước nới lỏng, do đó $d[v]$ không bị phép nới lỏng làm thay đổi.

---

[1] Có vẻ hơi lạ khi thuật ngữ "phép nới lỏng" được dùng cho một phép toán thắt chặt một cận trên. Việc dùng thuật ngữ này mang tính lịch sử. Kết quả của một bước nới lỏng có thể được xem như một phép nới lỏng sự ràng buộc $d[v] \le d[u] + w(u, v)$, mà, theo Bổ đề 25.3, phải được thỏa nếu $d[u] = \delta(s, u)$ và $d[v] = \delta(s, v)$. Nghĩa là, nếu $d[v] \le d[u] + w(u, v)$, ta không có "áp lực" thỏa sự ràng buộc này, do đó sự ràng buộc được "nới lỏng."

Mỗi thuật toán trong chương này gọi INITIALIZE-SINGLE-SOURCE và như vậy liên tục nới lỏng các cạnh. Hơn nữa, phép nới lỏng là biện pháp duy nhất qua đó các phần tử tiền vị và các ước lượng lộ trình ngắn nhất thay đổi. Các thuật toán trong chương này khác nhau về số lần mà chúng nới lỏng mỗi cạnh và thứ tự mà chúng nới lỏng các cạnh. Trong thuật toán Dijkstra và thuật toán các lộ trình ngắn nhất cho đồ thị phi chu trình có hướng, mỗi cạnh được nới lỏng chính xác một lần. Trong thuật toán Bellman-Ford, mỗi cạnh được nới lỏng vài lần.

Các tính chất của phép nới lỏng

Tính đúng đắn của các thuật toán trong chương này tùy thuộc vào các tính chất quan trọng của phép nới lỏng được tóm lược trong vài bổ đề kế tiếp. Hầu hết các bổ đề mô tả kết quả của việc thi hành một dãy các bước nới lỏng trên các cạnh của một đồ thị có hướng gia trọng đã được INITIALIZE-SINGLE-SOURCE khởi tạo. Ngoại trừ Bổ đề 25.9, các bổ đề này áp dụng cho mọi dãy các bước nới lỏng, chứ không chỉ cho các bước tạo ra các giá trị lộ trình ngắn nhất.

Bổ đề 25.4

Cho $G = (V, E)$ là một đồ thị có hướng gia trọng với hàm trọng số $w: E \rightarrow \mathbf{R}$, và cho $(u, v) \in E$. Thì, ngay sau khi nới lỏng cạnh (u, v) bằng cách thi hành RELAX(u,v,w), ta có $d[v] \leq d[u]+w(u,v)$.

Chứng minh Nếu, ngay trước khi nới lỏng cạnh (u, v), ta có $d[v] > d[u]+w(u, v)$, thì $d[v] = d[u]+w(u, v)$ sau đó. Nếu, thay vì, $d[v] \leq d[u]+w(u, v)$ ngay trước khi nới lỏng, thì cả $d[u]$ lẫn $d[v]$ đều không thay đổi, và do đó $d[v] \leq d[u] + w(u, v)$ sau đó.

Bổ đề 25.5

Cho $G = (V, E)$ là một đồ thị có hướng gia trọng với hàm trọng số $w: E \rightarrow \mathbf{R}$. Cho $s \in V$ là đỉnh nguồn, và cho đồ thị được khởi tạo bởi INITIALIZE-SINGLE-SOURCE (G, s). Thì, $d[v] \geq \delta(s, v)$ với tất cả $v \in V$, và bất biến này được duy trì trên bất kỳ dãy các bước nới lỏng nào trên các cạnh của G. Hơn nữa, một khi $d[v]$ đạt được cận dưới của nó $\delta(s, v)$, nó không bao giờ thay đổi.

Chứng minh Bất biến $d[v] \geq \delta(s, v)$ chắc chắn là đúng sau khi khởi tạo, bởi $d[s] = 0 \geq \delta(s, s)$ (lưu ý $\delta(s, s)$ là $-\infty$ nếu s nằm trên một chu trình trọng số âm hoặc bằng không là 0) và $d[v] = \infty$ hàm ý $d[v] \geq \delta(s, v)$ với tất cả $v \in V - \{s\}$. Ta sẽ dùng phép chứng minh bằng sự mâu thuẫn để chứng tỏ sự bất biến được duy trì trên mọi dãy các bước nới lỏng. Cho v là đỉnh đầu tiên mà một bước nới lỏng của một cạnh (u, v) khiến $d[v] < \delta(s, v)$. Thì, ngay sau khi nới lỏng cạnh (u, v), ta có

$$
\begin{aligned}
d[u] + w(u, v) &= d[v] \\
&< \delta(s, v) \\
&\leq \delta(s, u) + w(u, v) \qquad \text{(theo Bổ đề 25.3)},
\end{aligned}
$$

hàm ý rằng $d[u] < \delta(s, u)$. Nhưng do việc nới lỏng cạnh (u, v) không làm thay đổi $d[u]$, nên bất đẳng thức này phải là đúng ngay trước khi ta nới lỏng cạnh, điều này mâu thuẫn với sự chọn lựa v làm đỉnh đầu tiên mà $d[v] < \delta(s, v)$. Ta kết luận sự bất biến $d[v] \geq \delta(s, v)$ được duy trì với tất cả $v \in V$.

Để xem giá trị của $d[v]$ không bao giờ thay đổi một khi $d[v] = \delta(s, v)$, ta lưu ý khi đạt đến cận dưới củanó, $d[v]$ không thể giảm bởi ta vừa chứng tỏ rằng $d[v] \geq \delta(s,v)$, và nó không thể tăng bởi các bước nới lỏng không làm tăng các giá trị d.

Hệ luận 25.6

Giả sử rằng trong một đồ thị có hướng gia trọng $G = (V, E)$ với hàm trọng số $w : E \rightarrow \mathbf{R}$, không có lộ trình nào nối một đỉnh nguồn $s \in V$ với một đỉnh $v \in V$ đã cho. Vậy, sau khi đồ thị được INITIALIZE-SINGLE-SOURCE(G, s) khởi tạo, ta có $d[v] = \delta(s, v)$, và đẳng thức này được duy trì dưới dạng một bất biến đối với mọi dãy các bước nới lỏng trên các cạnh of G.

Chứng minh Theo Bổ đề 25.5, ta luôn có $\infty = \delta(s,v) \leq d[v]$; do đó, $d[v] = \infty = \delta(s, v)$.

Bổ đề dưới đây mang tính quyết định để chứng minh sự đúng đắn của các thuật toán các lộ trình ngắn nhất xuất hiện về sau trong chương này. Nó cung cấp đủ các điều kiện cho phương pháp nới lỏng để làm cho một ước lượng lộ trình ngắn nhất hội tụ đến một trọng số lộ trình ngắn nhất.

Bổ đề 25.7

Cho $G = (V, E)$ là một đồ thị có hướng gia trọng với hàm trọng số w: $E \rightarrow \mathbf{R}$, cho $s \in V$ là một đỉnh nguồn, và cho $s \rightsquigarrow u \rightarrow v$ là một lộ trình ngắn nhất trong G với vài đỉnh $u, v \in V$. Giả sử G được INITIALIZE-SINGLE-SOURCE(G, s) khởi tạo, và rồi một dãy các bước nới lỏng gộp lệnh gọi RELAX(u, v, w) được thi hành trên các cạnh của G. Nếu $d[u] = \delta(s, u)$ vào bất kỳ lúc nào trước lệnh gọi, thì $d[v] = \delta(s, v)$ mọi lúc sau lệnh gọi.

Chứng minh Theo Bổ đề 25.5, nếu $d[u] = \delta(s, u)$ tại một điểm nào đó trước khi nới lỏng cạnh (u, v), thì đẳng thức này đứng vững sau đó. Nói cụ thể, sau khi nới lỏng cạnh (u, v), ta có

$$
\begin{aligned}
d[v] &\le d[u] + w(u, v) && \text{(theo Bổ đề 25.4)} \\
&= \delta(s, u) + w(u, v) \\
&= \delta(s, v) && \text{(theo Hệ luận 25.2)} .
\end{aligned}
$$

Theo Bổ đề 25.5, $\delta(s, v)$ định cận $d[v]$ từ bên dưới, từ đó ta kết luận $d[v] = \delta(s, v)$, và đẳng thức này được duy trì sau đó.

Các cây các lộ trình ngắn nhất

Cho đến giờ, ta đã chứng tỏ phép nới lỏng khiến các ước lượng lộ trình ngắn nhất đi xuống một cách đơn điệu về phía các trọng số lộ trình ngắn nhất thực tế. Ta cũng muốn chứng tỏ rằng một khi một dãy các phép nới lỏng đã tính toán các trọng số lộ trình ngắn nhất thực tế, đồ thị con phần tử tiền vị G_π được cảm sinh bởi các giá trị π kết quả sẽ chính là một cây các lộ trình ngắn nhất của G. Ta bắt đầu với bổ đề dưới đây, chứng tỏ đồ thị con phần tử tiền vị luôn hình thành một cây có gốc mà gốc của nó là nguồn.

Bổ đề 25.8

Cho $G = (V, E)$ là một đồ thị có hướng gia trọng với hàm trọng số $w: E \to \mathbf{R}$ và đỉnh nguồn $s \in V$, và mặc nhận rằng G không chứa các chu trình trọng số âm khả dụng từ s. Như vậy, sau khi đồ thị được INITIALIZE-SINGLE-SOURCE(G, s) khởi tạo, đồ thị con phần tử tiền vị *Gn* hình thành một cây có gốc với gốc s, và bất kỳ dãy các bước nới lỏng nào trên các cạnh của G duy trì tính chất này dưới dạng một bất biến.

Chứng minh Thoạt đầu, đỉnh duy nhất trong G_π là đỉnh nguồn, và bổ đề tất nhiên là đúng. Xét một đồ thị con phần tử tiền vị G_π nảy sinh sau một dãy các bước nới lỏng. Trước tiên ta chứng minh G_π là phi chu trình. Do sự mâu thuẫn, ta giả sử rằng một bước nới lỏng nào đó tạo một chu trình trong đồ thị G_π. Cho chu trình là $c = \langle v_0, v_1, \ldots, v_k \rangle$, ở đó $v_k = v_0$. Như vậy, $\pi[v_i] = v_{i-1}$ với $i = 1, 2, \ldots, k$ và, không để mất tính tổng quát, ta có thể mặc nhận rằng chính phép nới lỏng của cạnh (v_{k-1}, v_k) đã tạo ra chu trình trong G_π.

Ta biện luận rằng tất cả các đỉnh trên chu trình c là khả dụng từ nguồn s. Tại sao? Mỗi đỉnh trên c có một phần tử tiền vị phi NIL, và do đó mỗi đỉnh trên c đã được gán một ước lượng lộ trình ngắn nhất hữu hạn khi nó được gán giá trị π phi NIL của nó. Theo Bổ đề 25.5, mỗi đỉnh trên chu trình c có một trọng số lộ trình ngắn nhất hữu hạn, hàm ý nó khả dụng từ s.

Ta sẽ xét các ước lượng lộ trình ngắn nhất trên c ngay trước lệnh gọi RELAX (v_{k-1}, v_k, w) và chứng tỏ c là một chu trình trọng số âm, do đó mâu thuẫn với giả thiết rằng G không chứa các chu trình trọng số âm

khả dụng từ nguồn. Ngay trước lệnh gọi, ta có $\pi[v_i] = v_{i-1}$ với $i = 1, 2,...,$ $k - 1$. Như vậy, với $i = 1, 2,..., k - 1$, lần cập nhật cuối cùng với $d[v_i]$ là do phép gán $d[v_i] \leftarrow d[v_{i-1}]+w(v_i, v_{i-1})$. Nếu $d[v_{i-1}]$ thay đổi từ đó, nó giảm. Do đó, ngay trước lệnh gọi RELAX(v_{k-1}, v_k, w), ta có

$$d[v_i] \geq d[v_{k-1}]+ w(v_{k-1}, v_i) \text{ với tất cả } i = 1, 2, ..., k-1 . \quad (25.1)$$

Do $\pi[v_k]$ được lệnh gọi thay đổi, nên ngay từ sớm ta cũng có bất đẳng thức ngặt

$$d[v_k] > d[v_{k-1}] + w(v_{k-1}, v_k) .$$

Cộng bất đẳng thức ngặt này với k -1 bất đẳng thức (25.1), ta được tổng các ước lượng lộ trình ngắn nhất quanh chu trình c:

$$\sum_{i=1}^{k} d[v_i] > \sum_{i=1}^{k} (d[v_{i-1}]+ w(v_{i-1}, v_i))$$
$$= \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i) .$$

Nhưng

$$\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d(v_{i-1}) ,$$

do mỗi đỉnh trong chu trình c xuất hiện chính xác một lần trong từng phép lấy tổng. Điều này hàm ý

$$0 > \sum_{i=1}^{k} w(v_{i-1}, v_i) .$$

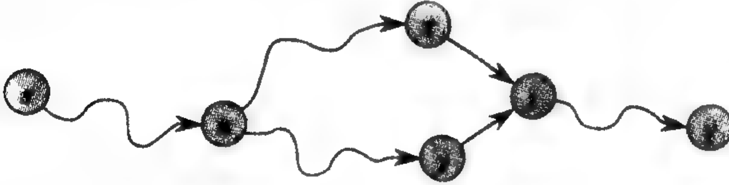
Như vậy, tổng các trọng số quanh chu trình c là âm, do đó ta có sự mâu thuẫn mong muốn.

Giờ đây ta đã chứng minh G_π là một đồ thị phi chu trình có hướng. Để chứng tỏ nó hình thành một cây có gốc với gốc s, đủ để chứng minh rằng (xem Bài tập 5.5-3) với mỗi đỉnh $v \in V_\pi$, ta có một lộ trình duy nhất từ s đến v trong G_π.

Đầu tiên ta phải chứng tỏ có một lộ trình từ s với mỗi đỉnh trong V_π. Các đỉnh trong V_π là những đỉnh có các giá trị π phi NIL, cộng với s. Ý tưởng ở đây là bằng phương pháp quy nạp ta chứng minh có một lộ trình tồn tại từ s đến tất cả các đỉnh trong V_π. Các chi tiết xin được dành lại làm Bài tập 25.1-6.

Để hoàn thành phần chứng minh của bổ đề, giờ đây ta phải chứng tỏ với bất kỳ đỉnh $v \in V_\pi$, ta có tối đa một lộ trình từ s đến v trong đồ thị G_π. Giả sử khác đi. Nghĩa là, giả sử có hai lộ trình đơn giản từ s đến một đỉnh v: p_1, có thể được phân tích thành $s \rightsquigarrow u \rightarrow x \rightsquigarrow z \rightsquigarrow v$, và p_2, mà ta

lại có thể phân tích thành $s \rightsquigarrow u \rightsquigarrow y \rightarrow z \rightsquigarrow v$, ở đó $x \neq y$. (Xem Hình 25.4.) Song như vậy, $\pi[z] = x$ và $\pi[z] = y$, hàm ý sự mâu thuẫn rằng $x = y$. Ta kết luận ở đó tồn tại một lộ trình đơn giản duy nhất trong G_π từ s đến v, và như vậy G_π hình thành một cây có gốc với gốc s.



Hình 25.4 Chứng tỏ một lộ trình trong G_π từ nguồn s đến đỉnh v là duy nhất. Nếu có hai lộ trình p_1 ($s \rightsquigarrow u \rightsquigarrow x \rightarrow z \rightsquigarrow v$) và p_2 ($s \rightsquigarrow u \rightsquigarrow y \rightarrow z \rightsquigarrow v$), ở đó $x \neq y$, thì $\pi[z] = x$ và $\pi[z] = y$, là một mâu thuẫn.

Giờ đây, ta có thể chứng tỏ nếu, sau khi đã thực hiện một dãy các bước nới lỏng, tất cả các đỉnh đã được gán các trọng số lộ trình ngắn nhất thực của chúng, như vậy đồ thị con phần tử tiền vị G_π là một cây các lộ trình ngắn nhất.

Bổ đề 25.9

Cho $G = (V, E)$ là một đồ thị có hướng gia trọng với hàm trọng số $w: E \rightarrow \mathbf{R}$ và đỉnh nguồn $s \in V$, và mặc nhận rằng G không chứa chu trình trọng số âm khả dụng từ s. Ta hãy gọi INITIALIZE-SINGLE-SOURCE(G, s) rồi thi hành bất kỳ dãy các bước nới lỏng nào trên các cạnh của G tạo ra $d[v] = \delta(s, v)$ với tất cả $v \in V$. Như vậy, đồ thị con phần tử tiền vị G_π là một cây các lộ trình ngắn nhất có gốc tại s.

Chứng minh Ta phải chứng minh ba tính chất của các cây các lộ trình ngắn nhất áp dụng cho G_π. Để nêu tính chất đầu tiên, ta phải chứng tỏ V_π là tập hợp các đỉnh khả dụng từ s. Theo định nghĩa, một trọng số lộ trình ngắn nhất $\delta(s, v)$ sẽ hữu hạn nếu và chỉ nếu v khả dụng từ s, và như vậy các đỉnh khả dụng từ s chính xác sẽ là những đỉnh có các giá trị d hữu hạn. Nhưng một đỉnh $v \in V - \{s\}$ đã được gán một giá trị hữu hạn cho $d[v]$ nếu và chỉ nếu $\pi[v] \neq$ NIL. Như vậy, các đỉnh trong V_π chính xác là các đỉnh khả dụng từ s.

Tính chất thứ hai trực tiếp do Bổ đề 25.8.

Do đó, phần còn lại đó là chứng minh tính chất cuối của các cây các lộ trình ngắn nhất: với tất cả $v \in V_\pi$, lộ trình đơn giản duy nhất $s \overset{p}{\rightsquigarrow} v$ trong G_π là một lộ trình ngắn nhất từ s đến v trong G. Cho $p = \langle v_0, v_1, ..., v_k \rangle$, ở đó $v_0 = s$ và $v_k = v$. For $i = 1, 2, ..., k$, ta có cả $d[v_i] = \delta(s, v_i)$ lẫn $d[v_i] \geq d[v_{i-1}] + w(v_{i-1}, v_i)$, từ đó ta kết luận $w(v_{i-1}, v_i) \leq \delta(s, v_i) - \delta(s, v_{i-1})$. Cộng các trọng số dọc theo lộ trình p sẽ cho ra

$$\begin{aligned} w(p) &> \sum_{i=1}^{k} w(v_{i-1}, v_i) \\ &\le \sum_{i=1}^{k} (\delta(s, v_i) - \delta(s, v_{i-1})) \\ &= \delta(s, v_k) - \delta(s, v_0) \\ &= \delta(s, v_k) . \end{aligned}$$

Dòng thứ ba xuất xứ từ tổng lồng gọn trên dòng thứ hai, và dòng thứ tư là do $\delta(s, v_0) = \delta(s, s) = 0$. Như vậy, $w(p) \le \delta(s, v_k)$. Do $\delta(s, v_k)$ là một cận dưới trên trọng số của bất kỳ lộ trình nào từ s đến v_k, nên ta kết luận rằng $w(p) = \delta(s, v_k)$, và như vậy p là một lộ trình ngắn nhất từ s đến $v = v_k$.

Bài tập

25.1-1

Nêu hai cây các lộ trình ngắn nhất cho đồ thị có hướng của Hình 25.2 ngoài hai cây đã nêu.

25.1-2

Nêu một ví dụ về một đồ thị có hướng gia trọng $G = (V, E)$ có hàm trọng số $w : E \to \mathbf{R}$ và nguồn s sao cho G thỏa tính chất sau đây: Với mọi cạnh $(u, v) \in E$, ta có một cây các lộ trình ngắn nhất có gốc tại s chứa (u, v) và một cây các lộ trình ngắn nhất khác có gốc tại s không chứa (u, v).

25.1-3

Tô điểm thêm phần chứng minh của Bổ đề 25.3 để điều quản các trường hợp ở đó các trọng số lộ trình ngắn nhất là ∞ hoặc $-\infty$.

25.1-4

Cho $G = (V, E)$ là một đồ thị có hướng gia trọng với đỉnh nguồn s, và cho G được khởi tạo bởi INITIALIZE-SINGLE-SOURCE(G, s). Chứng minh rằng nếu một dãy các bước nới lỏng ấn định $\pi[s]$ theo một giá trị phi NIL, thì G chứa một chu trình trọng số âm.

25.1-5

Cho $G = (V, E)$ là một đồ thị có hướng gia trọng không có các cạnh trọng số âm. Cho $s \in V$ là đỉnh nguồn, và ta hãy định nghĩa $\pi[v]$ như thường lệ: $\pi[v]$ là phần tử tiền vị của v trên vài lộ trình ngắn nhất đến v từ nguồn s nếu $v \in V - \{s\}$ khả dụng từ s, và bằng không là NIL. Nêu một ví dụ về một đồ thị G như vậy và một phép gán các giá trị π tạo ra một chu trình trong G_π. (Theo Bổ đề 25.8, một phép gán như vậy không

thể tạo bởi một dãy các bước nới lỏng.)

25.1-6

Cho $G = (V, E)$ là một đồ thị có hướng gia trọng với hàm trọng số $w : E \rightarrow \mathbf{R}$ và không có các chu trình trọng số âm. Cho $s \in V$ là đỉnh nguồn, và cho G được khởi tạo bởi INITIALIZE-SINGLE-SOURCE(G, s). Chứng minh rằng với mọi đỉnh $v \in V_\pi$, ở đó tồn tại một lộ trình từ s đến v trong G_π và tính chất này được duy trì dưới dạng một bất biến trên bất kỳ dãy phép nới lỏng nào.

25.1-7

Cho $G = (V, E)$ là một đồ thị có hướng gia trọng không chứa chu trình trọng số âm nào. Cho $s \in V$ là đỉnh nguồn, và cho G được khởi tạo bởi INITIALIZE-SINGLE-SOURCE(G, s). Chứng minh rằng có một dãy $|V| - 1$ bước nới lỏng tạo ra $d[v] = \delta(s, v)$ với tất cả $v \in V$.

25.1-8

Cho G là một đồ thị tùy ý có hướng gia trọng với một chu trình trọng số âm khả dụng từ đỉnh nguồn s. Chứng tỏ lúc nào cũng có thể kiến tạo một dãy vô hạn các phép nới lỏng của các cạnh của G, sao cho mọi phép nới lỏng sẽ làm thay đổi một ước lượng lộ trình ngắn nhất.

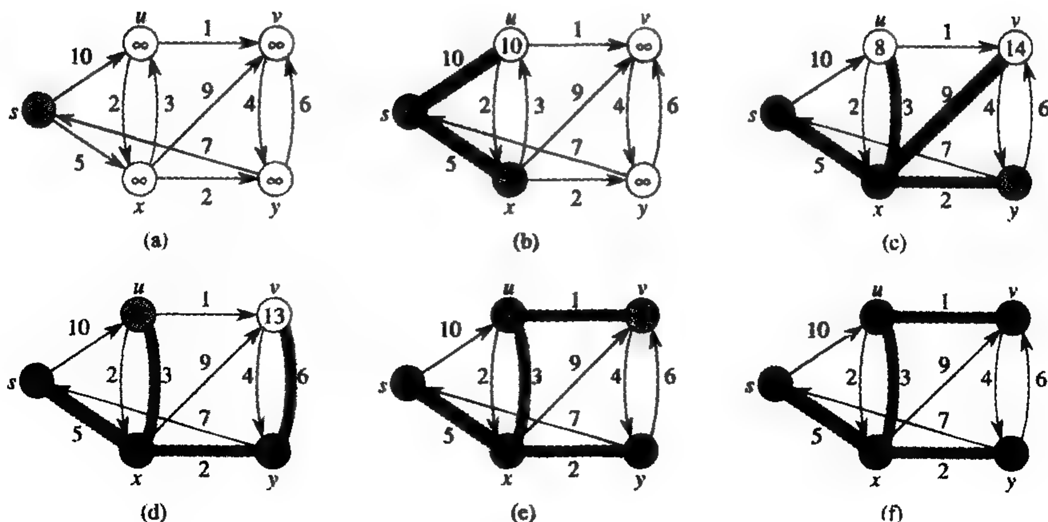
25.2 Thuật toán Dijkstra

Thuật toán Dijkstra giải quyết bài toán các lộ trình ngắn nhất nguồn-đơn trên một đồ thị có hướng gia trọng $G = (V, E)$ với trường hợp ở đó tất cả các trọng số cạnh là không âm. Do đó, trong đoạn này, ta mặc nhận rằng $w(u, v) \geq 0$ với mỗi cạnh $(u, v) \in E$.

Thuật toán Dijkstra duy trì một tập hợp S các đỉnh mà các trọng số lộ trình ngắn nhất của chúng từ nguồn s đã được xác định. Nghĩa là, với tất cả các đỉnh $v \in S$, ta có $d[v] = \delta(s, v)$. Thuật toán liên tục lựa đỉnh $u \in V - S$ với ước lượng lộ trình ngắn nhất cực tiểu, chèn u vào S, và nới lỏng tất cả các cạnh rời u. Trong kiểu thực thi dưới đây, ta duy trì một hàng đợi ưu tiên Q chứa tất cả các đỉnh trong $V - S$, được lập khóa bởi các giá trị d của chúng. Cách thực thi mặc nhận đồ thị G được biểu thị bởi các danh sách kề.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S ← Ø
```

```
3  Q ← V[G]
4  while Q ≠ ∅
5      do u ← EXTRACT-MIN(Q)
6          S ← S ∪ {u}
7          for mỗi đỉnh v ∈ Adj[u]
8              do RELAX (u, v, w)
```

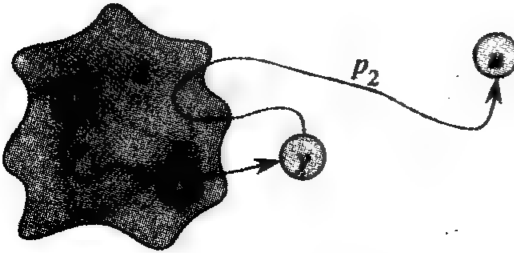


Hình 25.5 Thi hành thuật toán Dijkstra. Nguồn là đỉnh mút trái. Các ước lượng lộ trình ngắn nhất được nêu trong các đỉnh, và các cạnh tô bóng nêu rõ các giá trị phần tử tiền vị: nếu cạnh (u, v) được tô bóng, thì $\pi[v] = u$. Các đỉnh đen nằm trong tập hợp S, và các đỉnh trắng nằm trong hàng đợi ưu tiên $Q = V - S$. **(a)** Tình huống ngay trước lần lặp lại đầu tiên của vòng lặp **while** trong các dòng 4-8. Đỉnh tô bóng có giá trị d cực tiểu và được chọn làm đỉnh u trong dòng 5. **(b)-(f)** Tình huống sau mỗi lần lặp lại kế tiếp của vòng lặp **while**. Đỉnh tô bóng trong mỗi phần được chọn làm đỉnh u trong dòng 5 của lần lặp lại sau đó. Các giá trị d và π nêu trong phần (f) là các giá trị chung cuộc.

Thuật toán Dijkstra nới lỏng các cạnh như đã nêu trong Hình 25.5. Dòng 1 thực hiện phép khởi tạo thường lệ của các giá trị d và π, và dòng 2 khởi tạo tập hợp S theo tập hợp trống. Sau đó, dòng 3 khởi tạo hàng đợi ưu tiên Q để chứa tất cả các đỉnh trong $V - S = V - \emptyset = V$. Mỗi lần qua vòng lặp **while** của các dòng 4-8, một đỉnh u được trích từ $Q = V - S$ và được chèn vào tập hợp S. (Lần đầu tiên qua vòng lặp này, $u = s$.) Do đó, đỉnh u có ước lượng lộ trình ngắn nhất nhỏ nhất của bất kỳ đỉnh nào trong $V - S$. Sau đó, các dòng 7-8 nới lỏng mỗi cạnh (u, v) rời u, như vậy cập nhật ước lượng $d[v]$ và phần tử tiền vị $\pi[v]$ nếu có thể cải thiện

lộ trình ngắn nhất đến v bằng cách đi qua u. Nhận thấy các đỉnh không bao giờ được chèn vào Q sau dòng 3 và mỗi đỉnh được trích từ Q và được chèn vào S chính xác một lần, sao cho vòng lặp **while** của các dòng 4-8 lặp lại chính xác $|V|$ lần.

Bởi thuật toán Dijkstra luôn chọn đỉnh "nhạt nhất" hoặc "sát nhất" trong $V - S$ để chèn vào tập hợp S, ta nói rằng nó sử dụng một chiến lược tham. Các chiến lược tham được trình bày chi tiết trong Chương 17, nhưng bạn chẳng cần phải đọc chương đó để tìm hiểu thuật toán Dijkstra. Không phải lúc nào các chiến lược tham cũng mang lại các kết quả tối ưu nói chung, song như định lý dưới đây và hệ luận của nó cho thấy, thuật toán Dijkstra quả thực tính toán các lộ trình ngắn nhất. Vấn đề chính đó là chứng tỏ mỗi lần một đỉnh u được chèn vào tập hợp S, ta có $d[u] = \delta(s, u)$.



Hình 25.6 Phần chứng minh của Định lý 25.10. Tập hợp S không trống ngay trước khi đỉnh u được chèn vào nó. Một lộ trình ngắn nhất p từ nguồn s đến đỉnh u có thể được phân tích thành $s \overset{p}{\leadsto} x \to y \overset{p}{\leadsto} u$. ở đó y là đỉnh đầu tiên trên lộ trình không nằm trong $V - S$ và $x \in S$ đứng ngay trước y. Các đỉnh x và y là riêng biệt, nhưng ta có thể có $s = x$ hoặc $y = u$. Lộ trình p_2 có thể hoặc không thể nhập lại tập hợp S.

Định lý 25.10 (Tính đúng đắn của thuật toán Dijkstra)

Nếu ta chạy thuật toán Dijkstra trên một đồ thị có hướng gia trọng $G = (V, E)$ với hàm trọng số không âm w và nguồn s, thì vào lúc kết thúc, $d[u] = \delta(s, u)$ với tất cả các đỉnh $u \in V$.

Chứng minh Ta sẽ chứng tỏ với mỗi đỉnh $u \in V$, ta có $d[u] = \delta(s, u)$ vào lúc khi u được chèn vào tập hợp S và đẳng thức được duy trì sau đó.

Vì sự mâu thuẫn, cho u là đỉnh đầu tiên mà $d[u] \neq \delta(s, u)$ khi được chèn vào tập hợp S. Ta sẽ tập trung chú ý vào tình huống tại đầu lần lặp lại của vòng lặp **while** ở đó u được chèn vào S và suy ra sự mâu thuẫn rằng $d[u] = \delta(s, u)$ vào lúc đó bằng cách xét một lộ trình ngắn nhất từ s đến u. Ta phải có $u \neq s$ bởi vì s là đỉnh đầu tiên được chèn vào tập hợp S và $d[s] = \delta(s, s) = 0$ vào lúc đó. Bởi $u \neq s$, nên ta cũng có $S \neq 2$ ngay trước khi u được chèn vào S. Phải có một lộ trình nào đó từ s đến u,

bằng không với $d[u] = \delta(s, u) = \infty$ theo Hệ luận 25.6, sẽ vi phạm giả thiết của chúng ta rằng $d[u] \neq \delta(s,u)$. Bởi có ít nhất một lộ trình, nên ta có một lộ trình ngắn nhất p từ s đến u. Lộ trình p nối một đỉnh trong S, tức s, đến một đỉnh trong $V - S$, tức u. Ta hãy xét đỉnh đầu tiên y dọc theo p sao cho $y \in V - S$, và cho $x \in V$ là phần tử tiền vị của y. Như vậy, như đã nêu trong Hình 25.6, lộ trình p có thể được phân tích dưới dạng $s \overset{p_1}{\leadsto} x \to y \overset{p_2}{\leadsto} u$.

Ta biện luận rằng $d[y] = \delta(s, y)$ khi u được chèn vào S. Để chứng minh biện luận này, ta nhận thấy $x \in S$. Như vậy, do u được chọn làm đỉnh đầu tiên mà $d[u] \neq \delta(s, u)$ khi nó được chèn vào S, nên ta đã có $d[x] = \delta(s,x)$ khi x được chèn vào S. Cạnh (x, y) được nới lỏng vào lúc đó, do đó biện luận này là do Bổ đề 25.7.

Giờ đây, ta có thể được một mâu thuẫn để chứng minh định lý. Bởi y xảy ra trước u trên một lộ trình ngắn nhất từ s đến u và tất cả các trọng số cạnh đều không âm (nhất là trên lộ trình p_2), nên ta có $\delta(s, y) \leq \delta(s, u)$, và như vậy

$$\begin{aligned} d[y] &= \delta(s,y) \\ &\leq \delta(s, u) \qquad\qquad (25.2) \\ &\leq d[u] \qquad \text{(theo Bổ đề 25.5)}. \end{aligned}$$

Nhưng do cả đỉnh u lẫn y đều nằm trong $V - S$ khi u được chọn trong dòng 5, nên ta có $d[u] \leq d[y]$. Như vậy, hai bất đẳng thức trong (25.2) thực tế là các đẳng thức, cho ra

$$d[y] = \delta(s, y) = \delta(s, u) = d[u].$$

Bởi vậy, $d[u] = \delta(s, u)$, mâu thuẫn với chọn lựa u của chúng ta. Ta kết luận rằng vào lúc mỗi đỉnh $u \in V$ được chèn vào tập hợp S, ta có $d[u] = \delta(s, u)$, và theo Bổ đề 25.5, đẳng thức này vẫn đúng sau đó.

Hệ luận 25.11

Nếu ta chạy thuật toán Dijkstra trên một đồ thị có hướng gia trọng $G = (V, E)$ với hàm trọng số không âm w và nguồn s, thì vào lúc kết thúc, đồ thị con phần tử tiền vị G_π là một cây các lộ trình ngắn nhất có gốc tại s.

Chứng minh Tức thời từ Định lý 25.10 và Bổ đề 25.9.

Phân tích

Thuật toán Dijkstra nhanh tới mức nào? Xét trường hợp đầu tiên ở đó ta duy trì hàng đợi ưu tiên $Q = V - S$ dưới dạng một mảng tuyến tính. Với một kiểu thực thi như vậy, mỗi phép toán EXTRACT-MIN mất một

thời gian $O(V)$, và có $|V|$ phép toán như vậy, với một thời gian EXTRACT-MIN tổng cộng là $O(V^2)$. Mỗi đỉnh $v \in V$ được chèn vào tập hợp S chính xác một lần, do đó mỗi cạnh trong danh sách kề $Adj[v]$ được xem xét trong vòng lặp **for** của các dòng 4-8 chính xác một lần trong khi thi hành thuật toán. Do tổng các cạnh trong tất cả các danh sách kề là $|E|$, ta có một tổng $|E|$ lần lặp lại của vòng lặp **for** này, với mỗi lần lặp lại mất $O(1)$ thời gian. Như vậy, thời gian thực hiện của nguyên cả thuật toán là $O(V^2 + E) = O(V^2)$.

Tuy nhiên, nếu đồ thị là thưa, việc thực thi hàng đợi ưu tiên Q với một đống nhị phân là thực tiễn. Đôi lúc thuật toán kết quả được gọi là ***thuật toán Dijkstra đã sửa đổi***. Như vậy mỗi phép toán EXTRACT-MIN mất một thời gian $O(\lg V)$. Giống như trước, có $|V|$ phép toán như vậy. Thời gian để xây dựng đống nhị phân là $O(V)$. Phép gán $d[v] \leftarrow d[u] + w(u, v)$ trong RELAX được hoàn tất bởi lệnh gọi DECREASE-KEY(Q, v, $d[u] + w(u, v)$), mất một thời gian $O(\lg V)$ (xem Bài tập 7.5-4), và vẫn có tối đa $|E|$ phép toán như vậy. Do đó, tổng thời gian thực hiện là $O((V + E) \lg V)$, tức là $O(E \lg V)$ nếu tất cả các đỉnh là khả dụng từ nguồn.

Ta có thể thực tế đạt được một thời gian thực hiện là $O(V \lg V + E)$ bằng cách thực thi hàng đợi ưu tiên Q với một đống Fibonacci (xem Chương 21). Mức hao phí khấu trừ của mỗi trong số $|V|$ phép toán EXTRACT-MIN là $O(\lg V)$, và mỗi trong số $|E|$ lệnh gọi DECREASE-KEY chỉ mất $O(1)$ thời gian khấu trừ. Về mặt lịch sử, sự phát triển của các đống Fibonacci được thúc đẩy bởi việc quan sát thấy trong thuật toán Dijkstra đã sửa đổi, về tiềm năng, có nhiều lệnh gọi DECREASE-KEY hơn so với các lệnh gọi EXTRACT-MIN, do đó mọi phương pháp rút gọn thời gian khấu trừ của mỗi phép toán DECREASE-KEY theo $o(\lg V)$ mà không làm tăng thời gian khấu trừ của EXTRACT-MIN đều mang lại một cách thực thi nhanh hơn theo tiệm cận.

Thuật toán Dijkstra có vài điểm tương tự như thuật toán tìm kiếm độ rộng đầu tiên (xem Đoạn 23.2) và thuật toán Prim để tính toán các cây tỏa nhánh cực tiểu (xem Đoạn 24.2). Nó giống thuật toán tìm kiếm độ rộng đầu tiên ở chỗ tập hợp S tương ứng với tập hợp các đỉnh đen trong thuật toán tìm kiếm độ rộng đầu tiên; hết như các đỉnh trong S có các trọng số lộ trình ngắn nhất chung cuộc của chúng, các đỉnh đen trong một thuật toán tìm kiếm độ rộng đầu tiên cũng có các khoảng cách độ rộng đầu tiên đúng đắn của chúng. Thuật toán Dijkstra cũng giống như thuật toán Prim ở chỗ cả hai thuật toán đều dùng một hàng đợi ưu tiên để tìm đỉnh 'nhạt nhất" bên ngoài một tập hợp đã cho (tập hợp S trong thuật toán Dijkstra và cây đang được tăng trưởng trong thuật toán Prim), chèn đỉnh này vào tập hợp, và điều chỉnh một cách thích hợp các trọng

số của các đỉnh còn lại bên ngoài tập hợp.

Bài tập

25.2-1

Chạy thuật toán Dijkstra trên đồ thị có hướng của Hình 25.2, đầu tiên dùng đỉnh s làm nguồn rồi dùng đỉnh y làm nguồn. Trong kiểu dáng của Hình 25.5, nêu các giá trị d và π và các đỉnh trong tập hợp S sau mỗi lần lặp lại của vòng lặp **while**.

25.2-2

Nêu một ví dụ đơn giản của một đồ thị có hướng với các cạnh trọng số âm mà thuật toán Dijkstra tạo ra các đáp án sai. Tại sao phần chứng minh của Định lý 25.10 không thành công khi các cạnh trọng số âm được phép?

25.2-3

Giả sử ta thay đổi dòng 4 của thuật toán Dijkstra thành như sau.

4 **while** $|Q| > 1$

Thay đổi này khiến vòng lặp **while** thi hành $|V| - 1$ lần thay vì $|V|$ lần. Thuật toán đề nghị này có đúng không?

25.2-4

Ta có một đồ thị có hướng $G = (V, E)$ ở đó mỗi cạnh $(u, v) \in E$ có một giá trị kết hợp $r(u, v)$, là một số thực trong miền giá trị $0 \leq r(u, v) \leq 1$ biểu diễn cho độ tin cậy của một kênh truyền thông từ G đỉnh u đến đỉnh v. Ta diễn dịch $r(u, v)$ dưới dạng xác suất mà kênh từ u đến v sẽ không thất bại, và ta mặc nhận các xác suất này là độc lập. Nêu một thuật toán hiệu quả để tìm lộ trình đáng tin cậy nhất giữa hai đỉnh đã cho.

25.2-5

Cho $G = (V, E)$ là một đồ thị có hướng gia trọng, có hàm trọng số $w: E \rightarrow \{0, 1, ..., W - 1\}$ với một số nguyên không âm W. Sửa đổi thuật toán Dijkstra để tính toán các lộ trình ngắn nhất từ một đỉnh nguồn s đã cho trong $O(WV + E)$ thời gian.

25.2-6

Sửa đổi thuật toán của bạn từ Bài tập 25.2-5 để chạy trong $O((V + E) \lg W)$ thời gian. (*Mách nước.* Có thể có bao nhiêu ước lượng lộ trình ngắn nhất riêng biệt trong $V - S$ vào một điểm bất kỳ trong thời gian?)

25.3 Thuật toán Bellman-Ford

Thuật toán Bellman-Ford giải quyết bài toán các lộ trình ngắn nhất nguồn đơn trong trường hợp chung hơn ở đó các trọng số cạnh có thể là âm. Cho một đồ thị có hướng gia trọng $G = (V, E)$ với nguồn s và hàm trọng số $w : E \rightarrow \mathbf{R}$, thuật toán Bellman-Ford trả về một giá trị bool nêu rõ có hay không có một chu trình trọng số âm khả đụng từ nguồn. Nếu có một chu trình như vậy, thuật toán nêu rõ không có giải pháp nào tồn tại. Nếu không có một chu trình như vậy, thuật toán tạo ra các lộ trình ngắn nhất và các trọng số của chúng.

Cũng như thuật toán Dijkstra, thuật toán Bellman-Ford sử dụng kỹ thuật nới lỏng, từ từ giảm một ước lượng $d[v]$ trên trọng số của một lộ trình ngắn nhất từ nguồn s đến mỗi đỉnh $v \in V$ cho đến khi nó đạt được trọng số lộ trình ngắn nhất thực tế $\delta(s, v)$. Thuật toán trả về TRUE nếu và chỉ nếu đồ thị không chứa các chu trình trọng số âm khả đụng từ nguồn.

BELLMAN-FORD(G, w, s)

1 INITIALIZE-SINGLE-SOURCE(G, s)

2 **for** $i \leftarrow 1$ **to** $|V[G]| - 1$

3   **do for** mỗi cạnh $(u, v) \in E[G]$

4   **do** RELAX (u, v, w)

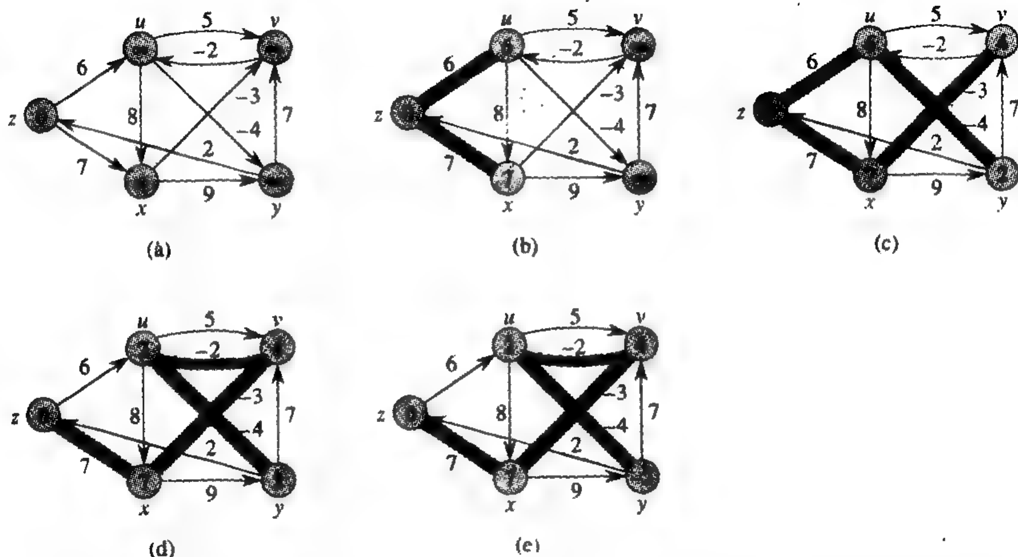
5 **for** mỗi cạnh $(u, v) \in E[G]$

6   **do if** $d[v] > d[u] + w(u, v)$

7   **then return** FALSE

8 **return** TRUE

Hình 25.7 nêu cách thi hành thuật toán Bellman-Ford trên một đồ thị có 5 đỉnh. Sau khi thực hiện phép khởi tạo thường lệ, thuật toán thực hiện $|V| - 1$ lượt trên các cạnh của đồ thị. Mỗi lượt là một lần lặp lại của vòng lặp **for** trong các dòng 2-4 và bao gồm phép nới lỏng mỗi lần một cạnh của đồ thị. Các Hình 25.7(b)-(e) nêu trạng thái của thuật toán sau mỗi trong số bốn lượt trên các cạnh. Sau khi thực hiện $|V| - 1$ lượt, các dòng 5-8 kiểm tra một chu trình trọng số âm và trả về giá trị Bool thích hợp. (Dưới đây ta sẽ thấy tại sao đợt kiểm tra này làm việc.)



Hình 25.7 Thi hành thuật toán Bellman-Ford. Nguồn là đỉnh z. Các giá trị d được nêu trong các đỉnh, và các cạnh tô bóng nêu rõ các giá trị π. Trong ví dụ cụ thể này, mỗi lượt sẽ nới lỏng các cạnh theo thứ tự từ điển: (u, v), (u, x), (u, y), (v, u), (x, v), (x, y), (y, v), (y, z), (z, u), (z, x). **(a)** Tình huống ngay trước lượt đầu tiên trên các cạnh. **(b)-(e)** Tình huống sau mỗi lượt sau đó trên các cạnh. Các giá trị d và π trong phần (e) là các giá trị chung cuộc. Thuật toán Bellman-Ford trả về TRUE trong ví dụ này.

Thuật toán Bellman-Ford chạy trong thời gian $O(VE)$, bởi phép khởi tạo trong dòng 1 mất $\Theta(V)$ thời gian, mỗi trong số $|V| - 1$ lượt trên các cạnh trong các dòng 2-4 mất $O(E)$ thời gian, và vòng lặp **for** của các dòng 5-7 mất $O(E)$ thời gian.

Để chứng minh tính đúng đắn của thuật toán Bellman-Ford, ta bắt đầu bằng cách chứng tỏ nếu không có các chu trình trọng số âm, thuật toán tính toán các trọng số lộ trình ngắn nhất đúng đắn với tất cả các đỉnh khả dụng từ nguồn. Phép chứng minh của bổ đề này chứa phần trực giác đàng sau thuật toán.

Bổ đề 25.12

Cho $G = (V, E)$ là một đồ thị có hướng gia trọng với nguồn s và hàm trọng số $w : E \rightarrow \mathbf{R}$, và mặc nhận G không chứa các chu trình trọng số âm khả dụng từ s. Như vậy, vào lúc kết thúc của BELLMAN-FORD, ta có $d[v] = \delta(s, v)$ với tất cả các đỉnh v khả dụng từ s.

Chứng minh Cho v là một đỉnh khả dụng từ s, và cho $p = \langle v_0, v_1, \ldots, v_k \rangle$ là một lộ trình ngắn nhất từ s đến v, ở đó $v_0 = s$ và $v_k = v$. Lộ trình p là đơn giản, và do đó $k \leq |V| - 1$. Bằng phương pháp quy nạp, ta muốn chứng minh rằng với $i = 0, 1, \ldots, k$, ta có $d[v_i] = \delta(s, v_i)$ sau lượt thứ i trên các cạnh của G và đẳng thức này được duy trì sau đó. Bởi ta có $|V| - 1$

lượt, nên biện luận này đủ để chứng minh bổ đề.

Về cơ bản, ta có $d[v_0] = \delta(s, v_0) = 0$ sau khi khởi tạo, và theo Bổ đề 25.5, đẳng thức này được duy trì sau đó.

Với bước quy nạp, ta mặc nhận $d[v_{i-1}] = \delta(s, v_{i-1})$ sau lượt thứ $(i - 1)$. Cạnh (v_{i-1}, v_i) được nới lỏng trong lượt thứ i, do đó theo Bổ đề 25.7, ta kết luận $d[v_i] = \delta(s, v_i)$ sau lượt thứ i và tại tất cả các lần sau đó, như vậy hoàn tất phần chứng minh.

Hệ luận 25.13

Cho $G = (V, E)$ là một đồ thị có hướng gia trọng với đỉnh nguồn s và hàm trọng số $w : E \rightarrow \mathbf{R}$. Thì với mỗi đỉnh $v \in V$, ta có một lộ trình từ s đến v nếu và chỉ nếu BELLMAN-FORD kết thúc với $d[v] < \infty$ khi nó chạy trên G.

Chứng minh Phần chứng minh tương tự như trường hợp Bổ đề 25.12 và được dùng làm Bài tập 25.3-2.

Định lý 25.14 (Tính đúng đắn của thuật toán Bellman-Ford)

Cho BELLMAN-FORD chạy trên một đồ thị có hướng gia trọng $G = (V, E)$ với nguồn s và hàm trọng số $w : E \rightarrow \mathbf{R}$. Nếu G không chứa các chu trình trọng số âm khả dụng từ s, thì thuật toán trả về TRUE, ta có $d[v] = \delta(s, v)$ với tất cả các đỉnh $v \in V$, và đồ thị con phần tử tiền vị G_π là một cây các lộ trình ngắn nhất có gốc tại s. Nếu G chứa một chu trình trọng số âm khả dụng từ S, thì thuật toán trả về FALSE.

Chứng minh Giả sử đồ thị G không chứa các chu trình trọng số âm khả dụng từ nguồn s. Trước tiên ta chứng minh biện luận cho rằng vào lúc kết thúc, $d[v] = \delta(s, v)$ với tất cả các đỉnh $v \in V$. Nếu đỉnh v khả dụng từ s, thì Bổ đề 25.12 sẽ chứng minh biện luận này. Nếu v không khả dụng từ s, thì biện luận là do Hệ luận 25.6. Như vậy, biện luận được chứng minh. Bổ đề 25.9, cùng với biện luận, hàm ý rằng G_π là một cây các lộ trình ngắn nhất. Giờ đây ta dùng biện luận để chứng tỏ BELLMAN-FORD trả về TRUE. Vào lúc kết thúc, ta có với tất cả các cạnh $(u, v) \in E$,

$$\begin{aligned} d[v] &= \delta(s,v) \\ &\leq \delta(s, u) + w(u, v) && \text{(theo Bổ đề 25.3)} \\ &= d[u] + w(u,v), \end{aligned}$$

và do đó không có đợt kiểm tra nào trong dòng 6 khiến BELLMAN-FORD trả về FALSE. Do đó nó trả về TRUE.

Ngược lại, giả sử rằng đồ thị G chứa một chu trình trọng số âm $c = \langle v_0, v_1, ..., v_k \rangle$, ở đó $v_0 = v_k$, là khả dụng từ nguồn s. Thì,

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0. \qquad (25.3)$$

Vì sự mâu thuẫn, ta mặc nhận rằng thuật toán Bellman-Ford trả về TRUE. Như vậy, $d[v_i] \le d[v_{i-1}] + w(v_{i-1}, v_i)$ với $i = 1, 2,..., k$. Tổng cộng các bất đẳng thức quanh chu trình c ta có

$$\sum_{i=1}^{k} d[v_i] \le \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

Như trong phần chứng minh của Bổ đề 25.8, mỗi đỉnh trong c xuất hiện chính xác một lần cho mỗi trong số hai phép lấy tổng đầu tiên. Như vậy,

$$\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}].$$

Hơn nữa, theo Hệ luận 25.13, $d[v_i]$ là hữu hạn với $i = 1, 2,..., k$. Như vậy,

$$0 \le \sum_{i=1}^{k} w(v_{i-1}, v_i),$$

là mâu thuẫn với bất đẳng thức (25.3). Ta kết luận thuật toán Bellman-Ford trả về TRUE nếu đồ thị G không chứa các chu trình trọng số âm khả dụng từ nguồn, và bằng không là FALSE.

Bài tập

25.3-1

Chạy thuật toán Bellman-Ford trên đồ thị có hướng của Hình 25.7, dùng đỉnh y làm nguồn. Nới lỏng các cạnh theo thứ tự từ điển trong mỗi lượt, và nêu các giá trị d và π sau mỗi lượt. Giờ đây, thay đổi trọng số của cạnh (y, v) thành 4 và chạy lại thuật toán, dùng z làm nguồn.

25.3-2

Chứng minh Hệ luận 25.13.

25.3-3

Cho một đồ thị có hướng gia trọng $G = (V, E)$ không có các chu trình trọng số âm, cho m là cực đại trên tất cả cặp đỉnh $u, v \in V$ của số cạnh cực tiểu trong một lộ trình ngắn nhất từ u đến v. (Ở đây, lộ trình ngắn nhất là theo trọng số, chứ không phải số lượng các cạnh.) Đề xuất một thay đổi đơn giản đối với thuật toán Bellman-Ford cho phép nó kết thúc trong $m + 1$ lượt.

25.3-4

Sửa đổi thuật toán Bellman-Ford sao cho nó ấn định $d[v]$ theo $-\infty$ với tất cả các đỉnh v mà ta có một chu trình trọng số âm trên vài lộ trình từ nguồn đến v.

25.3-5

Cho $G = (V, E)$ là một đồ thị có hướng gia trọng với hàm trọng số $w: E \to \mathbf{R}$. Nêu một thuật toán $O(VE)$ thời gian để, với mỗi đỉnh $v \in V$, tìm giá trị $\delta^*(v) = \min_{u \in V} \{\delta(u, v)\}$.

25.3-6*

Giả sử một đồ thị có hướng gia trọng $G = (V, E)$ có một chu trình trọng số âm. Nêu một thuật toán hiệu quả để liệt kê các đỉnh của một chu trình như vậy. Chứng minh thuật toán của bạn là đúng đắn.

25.4 Các lộ trình ngắn nhất nguồn đơn trong đồ thị phi chu trình có hướng

Nhờ nới lỏng các cạnh của một dag gia trọng (đồ thị phi chu trình có hướng) $G = (V, E)$ theo một đợt sắp xếp tôpô các đỉnh của nó, ta có thể tính toán các lộ trình ngắn nhất từ một nguồn đơn trong $\Theta(V + E)$ thời gian. Các lộ trình ngắn nhất luôn được định nghĩa tốt trong một dag, bởi cho dù có các cạnh trọng số âm, vẫn không thể có chu trình trọng số âm nào tồn tại.

Thuật toán bắt đầu bằng cách sắp xếp theo tôpô dag (xem Đoạn 23.4) để áp đặt một cách sắp xếp thứ tự tuyến tính trên các đỉnh. Nếu có một lộ trình từ đỉnh u đến đỉnh v, thì u đứng trước v trong đợt sắp xếp tôpô. Ta chỉ cần tiến hành một lượt trên các đỉnh theo thứ tự sắp xếp tôpô. Khi xử lý mỗi đỉnh, tất cả các cạnh rời đỉnh sẽ được nới lỏng.

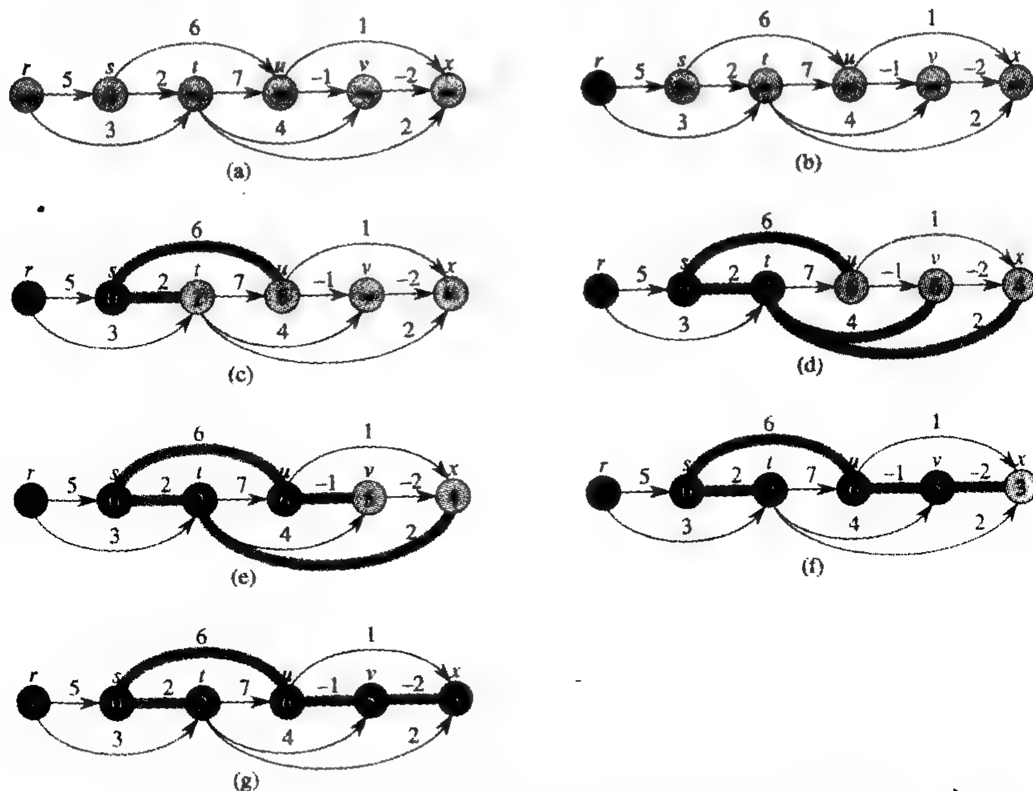
```
DAG-SHORTEST-PATHS (G, w, s)
1   sắp xếp tôpô các đỉnh của G
2   INITIALIZE-SINGLE-SOURCE(G, s)
3   for mỗi đỉnh u được lấy theo thứ tự sắp xếp tôpô
4           do for mỗi đỉnh v ∈ Adj[u]
5                   do RELAX(u, v, w)
```

Hình 25.8 có nêu một ví dụ về cách thi hành thuật toán này.

Thời gian thực hiện của thuật toán này được xác định bởi dòng 1 và bởi vòng lặp **for** của các dòng 3-5. Như đã nêu trong Đoạn 23.4, đợt sắp xếp tôpô có thể được thực hiện trong $\Theta(V + E)$ thời gian. Trong vòng

lặp **for** của các dòng 3-5, như trong thuật toán Dijkstra, ta có một lần lặp lại cho mỗi đỉnh. Với mỗi đỉnh, từng cạnh rời đỉnh được xem xét chính xác một lần. Tuy nhiên, khác với thuật toán Dijkstra, ta chỉ dùng $O(1)$ thời gian cho mỗi cạnh. Như vậy, thời gian thực hiện là $\Theta(V+E)$, là tuyến tính theo kích cỡ của một phép biểu diễn danh sách kề của đồ thị.

Định lý dưới đây chứng tỏ thủ tục DAG-SHORTEST-PATHS tính toán đúng đắn các lộ trình ngắn nhất.



Hình 25.8 Thi hành thuật toán cho các lộ trình ngắn nhất trong một đồ thị có hướng phi chu trình. Các đỉnh được sắp xếp theo tôpô từ trái qua phải. Đỉnh nguồn là s. Các giá trị d được nêu trong các đỉnh, và các cạnh tô bóng nêu rõ các giá trị π. **(a)** Tình huống trước lần lặp lại đầu tiên của vòng lặp **for** trong các dòng 3-5. **(b)-(g)** Tình huống sau mỗi lần lặp lại của vòng lặp **for** trong các dòng 3-5. Đỉnh mới được tô đen trong mỗi lần lặp lại sẽ được dùng làm v trong lần lặp lại đó. Các giá trị nêu trong phần (g) là các giá trị chung cuộc.

Định lý 25.15

Nếu một đồ thị có hướng gia trọng $G = (V, E)$ có đỉnh nguồn s và không có các chu trình, thì vào lúc kết thúc của thủ tục DAG-SHORTEST-PATHS, $d[v] = \delta(s, v)$ với tất cả các đỉnh $v \in V$, và đồ thị con phần tử tiền vị G_π là một cây các lộ trình ngắn nhất.

Chứng minh Trước tiên ta chứng tỏ $d[v] = \delta(s,v)$ với tất cả các đỉnh v

$\in V$ vào lúc kết thúc. Nếu v không khả dụng từ s, thì $d[v] = \delta(s,v) = \infty$ theo Hệ luận 25.6. Giờ đây, giả sử v là khả dụng từ s, sao cho có một lộ trình ngắn nhất $p = \langle v_0, v_1, ..., v_k \rangle$, ở đó $v_0 = s$ và $v_k = v$. Bởi vì ta xử lý các đỉnh theo thứ tự sắp xếp tôpô, các cạnh trên p được nới lỏng theo thứ tự $(v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k)$. Một phương pháp quy nạp đơn giản dùng Bổ đề 25.7 (như trong phần chứng minh của Bổ đề 25.12) chứng tỏ $d[v_i] = \delta(s, v_i)$ vào lúc kết thúc với $i = 0, 1, ..., k$. Cuối cùng, theo Bổ đề 25.9, G_π là một cây các lộ trình ngắn nhất.

Có thể áp dụng thuật toán này vào việc xác định các lộ trình tới hạn trong phân tích ***biểu đồ PERT*** [2]. Các cạnh biểu diễn các công việc được thực hiện, và các trọng số cạnh biểu diễn các thời gian cần thiết để thực hiện các công việc cụ thể. Nếu cạnh (u, v) nhập đỉnh v và cạnh (v, x) rời v, thì công việc (u, v) phải được thực hiện trước công việc (v, x). Một lộ trình qua dag này biểu diễn một dãy công việc phải được thực hiện theo một thứ tự cụ thể. Một ***lộ trình tới hạn*** là một lộ trình *dài nhất* qua dag, tương ứng với thời gian dài nhất để thực hiện một dãy các công việc có sắp xếp thứ tự. Trọng số của một lộ trình tới hạn là một cận dưới trên tổng thời gian để thực hiện tất cả mọi công việc. Ta có thể tìm một lộ trình tới hạn bằng cách

- phủ định các trọng số cạnh và chạy DAG-SHORTEST-PATHS, hoặc
- chạy DAG-SHORTEST-PATHS, thay "∞" bằng "- ∞" trong dòng 2 của INITIALIZE-SINGLE-SOURCE và ">" bằng "<" trong thủ tục RELAX.

Bài tập

25.4-1

Chạy DAG-SHORTEST-PATHS trên đồ thị có hướng của Hình 25.8, dùng đỉnh r làm nguồn.

25.4-2

Giả sử ta thay đổi dòng 3 của DAG-SHORTEST-PATHS thành

3 **for** $|V| - 1$ đỉnh đầu tiên, được lấy theo thứ tự sắp xếp tôpô

Chứng tỏ thủ tục vẫn đúng.

25.4-3

Cách trình bày biểu đồ PERT nêu trên đây có phần thiếu tự nhiên. Sẽ là tự nhiên hơn nêu các đỉnh biểu diễn các công việc và các cạnh biểu diễn các hạn chế kiểm tra trình tự; nghĩa là, cạnh (u, v) sẽ nêu rõ

---

[2] "PERT" là tên tắt của "program evaluation and review technique."

công việc *u* phải được thực hiện trước công việc *v*. Như vậy, các trọng số được gán cho các đỉnh, chứ không phải các cạnh. Sửa đổi thủ tục DAG-SHORTEST-PATHS sao cho nó tìm một lộ trình dài nhất trong một đồ thị phi chu trình có hướng với các đỉnh gia trọng trong thời gian tuyến tính.

25.4-4

Nêu một thuật toán hiệu quả để đếm tổng các lộ trình trong một đồ thị phi chu trình có hướng. Phân tích thuật toán của bạn và chú giải về tính thực tiễn của nó.

25.5 Các hạn chế sai phân và các lộ trình ngắn nhất

Trong bài toán lập trình tuyến tính chung, ta muốn tối ưu hóa một hàm tuyến tính tùy thuộc vào một tập hợp các bất đẳng thức tuyến tính. Trong đoạn này, ta nghiên cứu một trường hợp đặc biệt của lập trình tuyến tính có thể rút gọn vào việc tìm các lộ trình ngắn nhất từ một nguồn đơn. Sau đó, có thể giải quyết bài toán các lộ trình ngắn nhất nguồnđơn kết quả bằng thuật toán Bellman-Ford, và do đó cũng giải quyết bài toán lập trình tuyến tính.

Lập trình tuyến tính

Trong ***bài toán lập trình tuyến tính*** chung, ta có một ma trận $m \times n$ A, một vectơ-m b, và một vectơ-n c. Ta muốn tìm một vector x gồm n thành phần tối đa hóa ***hàm mục tiêu*** $\sum_{i=1}^{n} c_i x_i$ lệ thuộc vào các hạn chế m mà $Ax \leq b$ cung cấp.

Có thể diễn tả nhiều bài toán dưới dạng các chương trình tuyến tính, và vì lý do này phần lớn công trình đã xoáy vào các thuật toán lập trình tuyến tính. ***Thuật toán đơn hình*** [simplex algorithm][3] giải quyết các

---

[3] Thuật toán đơn hình tìm một giải pháp tối ưu cho một bài toán lập trình tuyến tính bằng cách xét một dãy các điểm trong vùng khả thi—vùng trong không gian *n* thỏa $Ax \leq b$. Thuật toán này dựa trên sự việc đó là một giải pháp tối đa hóa hàm mục tiêu trên vùng khả thi xảy ra tại một "cực điểm," hoặc "góc," của vùng khả thi. Thuật toán đơn hình tiến hành từ góc này sang góc khác của vùng khả thi cho đến khi không thể có sự cải thiện nào thêm của hàm mục tiêu. Một "đơn hình" là bao lồi (xem Đoạn 35.3) của $d + 1$ điểm trong không gian chiều d (như một tam giác trong mặt phẳng, hoặc một tứ diện trong không gian-3). Theo Dantzig [53], ta có thể xem phép toán di chuyển từ góc này sang góc khác dưới dạng một phép toán trên một đơn hình phái sinh từ một phép diễn dịch "đôi" của bài toán lập trình tuyến tính—từ đó có tên "phương pháp đơn hình."

chương trình tuyến tính chung rất nhanh trong thực tế. Tuy nhiên, với vài nhập liệu được trù tính cẩn thận, phương pháp đơn hình có thể yêu cầu thời gian mũ. Có thể giải quyết các chương trình tuyến tính chung trong thời gian đa thức bằng ***thuật toán ellipsoid***, chạy chậm trong thực tế, hoặc ***thuật toán Karmarkar***, mà trong thực tế thường cạnh tranh với phương pháp đơn hình.

Do việc đầu tư toán học cần hiểu rõ và phân tích chúng, nên tài liệu này không đề cập các thuật toán lập trình tuyến tính chung. Tuy nhiên, với vài lý do, ta cần hiểu rõ cách xác lập các bài toán lập trình tuyến tính. Trước tiên, biết được có thể "áp đổi" [cast] một bài toán đã cho dưới dạng một bài toán lập trình tuyến tính có quy mô đa thức, ta biết ngay sẽ có một thuật toán thời gian đa thức cho bài toán. Thứ hai, có nhiều trường hợp đặc biệt của lập trình tuyến tính ở đó tồn tại các thuật toán nhanh hơn. Ví dụ, như đã nêu trong đoạn này, bài toán các lộ trình ngắn nhất nguồn đơn là một trường hợp đặc biệt của lập trình tuyến tính. Các bài toán khác có thể được áp đổi dưới dạng lập trình tuyến tính sẽ gộp bài toán lộ trình ngắn nhất cặp đơn (Bài tập 25.5-4) và bài toán luồng cực đại (Bài tập 27.1-8).

Đôi lúc ta không thực sự quan tâm về hàm mục tiêu; ta chỉ muốn tìm một ***giải pháp khả thi*** bất kỳ, nghĩa là, bất kỳ vectơ x nào thỏa $Ax \leq b$, hoặc để xác định không có giải pháp khả thi nào tồn tại. Ta sẽ tập trung vào một ***bài toán về tính khả thi*** như vậy.

Các hệ thống của các hạn chế sai phân

Trong một hệ thống các hạn chế sai phân, mỗi hàng của ma trận lập trình tuyến tính A chứa một 1 và một -1, và tất cả các khoản nhập khác của A là 0. Như vậy, các hạn chế do $Ax \leq b$ là một tập hợp m hạn chế sai phân bao hàm n ẩn số, ở đó mỗi ràng buộc là một bất đẳng thức tuyến tính đơn giản có dạng

$$x_j - x_i \leq b_k,$$

ở đó $1 \leq i, j \leq n$ và $1 \leq k \leq m$.

Ví dụ, xét bài toán tìm vector-5 $x = (x_i)$ thỏa

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} \leq \begin{pmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{pmatrix}$$

Bài toán này tương đương với việc tìm các ẩn số x_i, với i sao cho the following 8 các hạn chế sai phân được thỏa:

$$
\begin{aligned}
x_1 - x_2 &\leq 0, \\
x_1 - x_5 &\leq -1, \\
x_2 - x_5 &\leq 1, \\
x_3 - x_1 &\leq 5, \\
x_4 - x_1 &\leq 4, \\
x_4 - x_3 &\leq -1, \\
x_5 - x_3 &\leq -3, \\
x_5 - x_4 &\leq -3,
\end{aligned}
\tag{25.4}
$$

Một giải pháp cho bài toán này là $x = (-5, -3, 0, -1, -4)$, như có thể xác minh trực tiếp bằng cách kiểm tra từng bất đẳng thức. Thực vậy, có nhiều giải pháp cho bài toán này. Một giải pháp khác đó là $x' = (0, 2, 5, 4, 1)$. Hai giải pháp này có liên quan: mỗi thành phần của x' lớn hơn 5 so với thành phần tương ứng của x. Sự việc này không đơn thuần là sự trùng hợp ngẫu nhiên.

Bổ đề 25.16

Cho $x = (x_1, x_2, \ldots, x_n)$ là một giải pháp cho một hệ thống $Ax \leq b$ các hạn chế sai phân, và cho d là một hằng bất kỳ. Thì $x + d = (x_1 + d, x_2 + d, \ldots, x_n + d)$ cũng là một giải pháp cho $Ax \leq b$.

Chứng minh Với mỗi x_i và x_j, ta có $(x_j + d) - (x_i + d) = x_j - x_i$. Như vậy, nếu x thỏa $Ax \leq b$, thì $x + d$ cũng vậy.

Các hệ thống của các hạn chế sai phân xảy ra trong nhiều ứng dụng khác nhau. Ví dụ, các ẩn số x_i có thể là số lần ở đó các sự kiện xảy ra. Mỗi ràng buộc có thể được xem như phát biểu rằng một sự kiện không thể xảy ra quá trễ so với một sự kiện khác. Có lẽ các sự kiện là các công việc sẽ được thực hiện trong khi xây dựng một căn nhà. Nếu việc đào móng bắt đầu vào lúc x_1 và mất 3 ngày và việc đổ bê tông cho móng bắt đầu vào lúc x_2, ta có thể đề nghị $x_2 \geq x_1 + 3$ hoặc, tương đương, $x_1 - x_2 \leq -3$. Như vậy, sự ràng buộc tính giờ tương đối có thể được diễn tả dưới dạng một ràng buộc sai phân.

Các đồ thị ràng buộc

Quả có lợi khi diễn dịch các hệ thống của các hạn chế sai phân theo quan điểm lý thuyết đồ thị. Ý tưởng đó là trong một hệ thống $Ax \leq b$ các hạn chế sai phân, ma trận lập trình tuyến tính $n \times m$ A có thể được xem như một ma trận liên thuộc (xem Bài tập 23.1-7) cho một đồ thị có n đỉnh và m cạnh. Mỗi đỉnh v_i trong đồ thị, với $i = 1, 2, \ldots, n$, tương ứng với một trong số n ẩn biến x_i. Mỗi cạnh có hướng trong đồ thị tương ứng với một trong m bất đẳng thức bao hàm hai ẩn số.

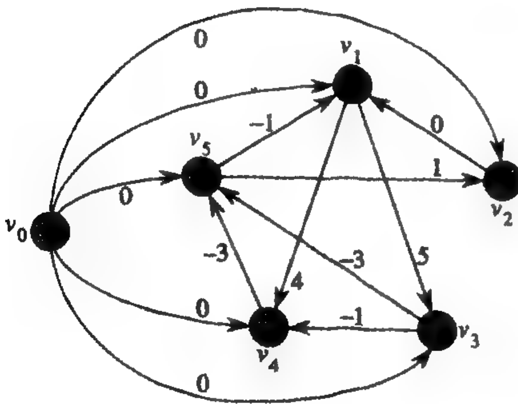
Chính thức hơn, cho một hệ thống $Ax \leq b$ các hạn chế sai phân, ***đồ thị ràng buộc*** [constraint graph] tương ứng là một đồ thị có hướng gia trọng $G = (V, E)$, ở đó

$$V = \{v_0, v_1, \ldots, v_n\}$$

và

$$E = \{(v_i, v_j) : x_j - x_i \leq b_k \text{ là một ràng buộc}\}$$
$$\cup\{(v_0, v_1), (v_0, v_2), (v_0, v_3), \ldots, (v_0, v_n)\} .$$

Như sẽ thấy dưới đây, đỉnh bổ sung v_0 được sát nhập để bảo đảm rằng mọi đỉnh khác khả đụng từ nó. Như vậy, tập hợp đỉnh V bao gồm một đỉnh v_i với mỗi ẩn số x_i, cộng với một đỉnh bổ sung v_0. Tập hợp cạnh E chứa một cạnh cho mỗi ràng buộc sai phân, cộng với một cạnh (v_0, v_i) cho mỗi ẩn số x_i. Nếu $x_j - x_i \leq b_k$ là một ràng buộc sai phân, thì trọng số của cạnh (v_i, v_j) là $w(v_i, v_j) = b_k$. Trọng số của mỗi cạnh rời v_0 là 0. Hình 25.9 nêu đồ thị ràng buộc cho hệ thống (25.4) các hạn chế sai phân.



Hình 25.9 Đồ thị ràng buộc tương ứng với hệ thống (25.4) các hạn chế sai phân. Giá trị của $\delta(v_0, v_i)$ được nêu trong mỗi đỉnh v_i. Một giải pháp khả thi cho hệ thống là $x = (-5, -3, 0, -1, -4)$.

Định lý dưới đây chứng tỏ có thể có một giải pháp cho một hệ thống các hạn chế sai phân bằng cách tìm các trọng số lộ trình ngắn nhất trong đồ thị ràng buộc tương ứng.

Định lý 25.17

Cho của hệ thống $Ax \leq b$ các hạn chế sai phân, cho $G = (V, E)$ là đồ thị ràng buộc tương ứng. Nếu G không chứa các chu trình trọng số âm, thì

$$x = (\delta(v_0, v_1), \delta(v_0, v_2), \delta(v_0, v_3), \ldots, \delta(v_0, v_n)) \qquad (25.5)$$

là một giải pháp khả thi cho hệ thống. Nếu G chứa một chu trình trọng số âm, thì không có giải pháp khả thi nào cho hệ thống.

Chứng minh Trước tiên ta chứng tỏ nếu đồ thị ràng buộc không chứa các chu trình trọng số âm, thì phương trình (25.5) cho ra một giải pháp khả thi. Xét bất kỳ cạnh $(v_i, v_j) \in E$. Theo Bổ đề 25.3, $\delta(v_0, v_j) \leq \delta(v_0, v_i) + w(v_i, v_j)$ hoặc, tương đương, $\delta(v_0, v_j) - \delta(v_0, v_i) \leq w(v_i, v_j)$. Như vậy, việc cho $x_i = \delta(v_0, v_i)$ và $x_j = \delta(v_0, v_j)$ sẽ thỏa sự ràng buộc sai phân $x_j - x_i \leq w(v_i, v_j)$ tương ứng với cạnh (v_i, v_j).

Giờ đây ta chứng tỏ nếu đồ thị ràng buộc chứa một chu trình trọng số âm, thì hệ thống các hạn chế sai phân không có giải pháp khả thi. Không để mất tính tổng quát, cho chu trình trọng số âm là $c = \langle v_1, v_2, \ldots, v_k \rangle$, ở đó $v_1 = v_k$. (Đỉnh v_0 không thể nằm trên chu trình c, bởi nó không có các cạnh vào.) Chu trình c tương ứng với các hạn chế sai phân dưới đây:

$$\begin{aligned}
x_2 - x_1 &\leq w(v_1, v_2), \\
x_3 - x_2 &\leq w(v_2, v_3), \\
&\vdots \\
x_k - x_{k-1} &\leq w(v_{k-1}, v_k), \\
x_1 - x_k &\leq w(v_k, v_1).
\end{aligned}$$

Do bất kỳ giải pháp nào cho x đều phải thỏa mỗi trong số k bất đẳng thức này, nên bất kỳ giải pháp nào cũng phải thỏa bất đẳng thức là kết quả khi ta tổng cộng chúng với nhau. Nếu ta tổng cộng các cạnh bên trái, từng ẩn số x_i được bổ sung trong một lần và được trừ đi một lần, sao cho cạnh bên trái của tổng là 0. Cạnh tay phải cộng vào $w(c)$, và như vậy ta được $0 \leq w(c)$. Nhưng do c là một chu trình trọng số âm, $w(c) < 0$, và do đó mọi giải pháp cho x đều phải thỏa $0 \leq w(c) < 0$, là không thể.

Giải quyết các hệ thống của các hạn chế sai phân

Định lý 25.17 cho biết ta có thể dùng thuật toán Bellman-Ford để giải quyết một hệ thống các hạn chế sai phân. Do có các cạnh từ đỉnh nguồn v_0 đến tất cả các đỉnh khác trong đồ thị ràng buộc, mọi chu trình trọng số âm trong đồ thị ràng buộc là khả dụng từ v_0. Nếu thuật toán Bellman-Ford trả về TRUE, thì các trọng số lộ trình ngắn nhất cho ra một giải pháp khả thi cho hệ thống. Ví dụ, trong Hình 25.9, các trọng số lộ trình ngắn nhất cung cấp giải pháp khả thi $x = (-5, -3, 0, -1, -4)$, và theo Bổ đề 25.16, $x = (d - 5, d - 3, d, d - 1, d - 4)$ cũng là một giải pháp khả thi với một hằng d. Nếu thuật toán Bellman-Ford trả về FALSE, ta không có giải pháp khả thi nào cho hệ thống các hạn chế sai phân.

Một hệ thống các hạn chế sai phân với m hạn chế trên n ẩn số sẽ tạo ra một đồ thị với $n + 1$ đỉnh và $n + m$ cạnh. Như vậy, dùng thuật toán Bellman-Ford, ta có thể giải quyết hệ thống trong $O((n + 1)(n + m)) = O(n^2 + nm)$ thời gian. Bài tập 25.5-5 yêu cầu bạn chứng tỏ thuật toán thực tế chạy trong $O(nm)$ thời gian, cho dù m nhỏ hơn nhiều so với n.

Bài tập

25.5-1

Tìm một giải pháp khả thi hoặc xác định không có giải pháp khả thi nào tồn tại cho hệ thống các hạn chế sai phân dưới đây:

$x_1 - x_2 \leq 1$,

$x_1 - x_4 \leq -4$,

$x_2 - x_3 \leq 2$,

$x_2 - x_5 \leq 7$,

$x_2 - x_6 \leq 5$,

$x_3 - x_6 \leq 10$,

$x_4 - x_2 \leq 2$,

$x_5 - x_1 \leq -1$,

$x_5 - x_4 \leq 3$,

$x_6 - x_3 \leq -8$.

25.5-2

Tìm một giải pháp khả thi hoặc xác định không có giải pháp khả thi nào tồn tại cho hệ thống các hạn chế sai phân dưới đây:

$x_1 - x_2 \leq 4$,

$x_1 - x_5 \leq 5$,

$x_2 - x_4 \leq -6$,

$x_3 - x_2 \leq 1$,

$x_4 - x_1 \leq 3$,

$x_4 - x_3 \leq 5$,

$x_4 - x_5 \leq 10$,

$x_5 - x_3 \leq -4$,

$x_5 - x_4 \leq -8$.

25.5-3

Có thể có trọng số lộ trình ngắn nhất nào từ đỉnh mới v_0 trong một đồ thị ràng buộc là dương hay không? Giải thích.

25.5-4

Biểu diễn bài toán lộ trình ngắn nhất cặp đơn dưới dạng một chương trình tuyến tính.

25.5-5

Nêu cách sửa đổi sơ thuật toán Bellman-Ford sao cho khi dùng nó để giải quyết một hệ thống các hạn chế sai phân với *m* bất đẳng thức trên *n* ẩn số, thời gian thực hiện là $O(nm)$.

25.5-6

Nêu cách giải quyết một hệ thống các hạn chế sai phân bằng một thuật toán tương tự Bellman-Ford chạy trên một đồ thị ràng buộc mà không cần thêm đỉnh v_0.

25.5-7 *

Cho $Ax \leq b$ là một hệ thống *m* hạn chế sai phân trong *n* ẩn số. Chứng tỏ thuật toán Bellman-Ford, khi chạy trên đồ thị ràng buộc tương ứng, sẽ tối đa hóa $\sum_{i=1}^{n} x_i$ lệ thuộc vào $Ax \leq b$ và $x_i \leq 0$ với tất cả x_i.

25.5-8

Chứng tỏ thuật toán Bellman-Ford, khi chạy trên đồ thị ràng buộc cho một hệ thống $Ax \leq b$ các hạn chế sai phân, sẽ giảm thiểu số lần $(\max\{x_i\} - \min\{x_i\})$ lệ thuộc vào $Ax \leq b$. Giải thích sự việc này thích hợp như thế nào nếu thuật toán được dùng để lên lịch các công việc xây dựng.

25.5-9

Giả sử rằng mọi hàng trong ma trận A của một chương trình tuyến tính $Ax \leq b$ tương ứng với một ràng buộc sai phân, một ràng buộc biến đơn có dạng $x_i \leq b_k$, hoặc một ràng buộc biến đơn có dạng $-x_i \leq b_k$. Nêu cách thích ứng thuật toán Bellman-Ford để giải quyết sự đa dạng này của hệ thống ràng buộc.

25.5-10

Giả sử rằng ngoài hệ thống các hạn chế sai phân, ta muốn điều quản các hạn chế đẳng thức có dạng $x_i = x_j + b_k$. Nêu cách thích ứng thuật toán Bellman-Ford để giải quyết sự đa dạng này của hệ thống ràng buộc.

25.5-11

Nêu một thuật toán hiệu quả để giải quyết một hệ thống $Ax \leq b$ các hạn chế sai phân khi tất cả các thành phần của b mang giá trị thực và tất cả các ẩn số x_i phải là các số nguyên.

25.5-12*

Nêu thuật toán hiệu quả để giải quyết một hệ thống $Ax \leq b$ các hạn chế sai phân khi tất cả các thành phần của b mang giá trị thực và một vài, nhưng không nhất thiết là tất cả, ẩn số x_i phải là các số nguyên.

Các Bài Toán

25-1 cách cải thiện Bellman-Ford của Yen

Giả sử ta sắp xếp các phép nới lỏng cạnh trong mỗi lượt của thuật toán Bellman-Ford như sau. Trước lượt đầu tiên, ta gán một thứ tự tuyến tính tùy ý $v_1, v_2, \ldots, v_{|V|}$ cho các đỉnh của đồ thị nhập liệu $G = (V, E)$. Sau đó, ta phân hoạch tập hợp cạnh E thành $E_f \cup E_b$, ở đó $E_f = \{(v_i, v_j) \in E : i < j\}$ và $E_b = \{(v_i, v_j) \in E : i > j\}$. Định nghĩa $G_f = (V, E_f)$ và $G_b = (V, E_b)$.

a. Chứng minh G_f là phi chu trình với kiểu sắp xếp tôpô $\langle v_1, v_2, \ldots, v_{|V|}\rangle$ và G_b là phi chu trình với kiểu sắp xếp tôpô $\langle v_{|V|}, v_{|V|-1}, \ldots, v_1\rangle$.

Giả sử ta thực thi mỗi lượt thuật toán Bellman-Ford theo cách dưới đây. Ta ghé mỗi đỉnh theo thứ tự $v_1, v_2, \ldots, v_{|V|}$, nới lỏng các cạnh của E_f rời đỉnh. Sau đó, ta ghé mỗi đỉnh theo thứ tự $v_{|V|}, v_{|V|-1}, \ldots, v_1$, nới lỏng các cạnh của E_b rời đỉnh.

b. Chứng minh với lược đồ này, nếu G không chứa các chu trình

trọng số âm khả dụng từ đỉnh nguồn s, thì sau chỉ $\lceil |V| /2 \rceil$ lượt trên các cạnh, $d[v] = \delta(s, v)$ với tất cả các đỉnh $v \in V$.

c. Lược đồ này ảnh hưởng đến thời gian thực hiện của thuật toán Bellman-Ford ra sao?

25-2 Lồng ghép các hộp

Một hộp chiều d có các chiều $(x_1, x_2,..., x_d)$ ***lồng ghép*** trong một hộp khác có các chiều $(y_1, y_2,...,y_d)$ nếu ở đó tồn tại một phép hoán vị π trên $\{1, 2,..., d\}$ sao cho $x_{\pi(1)} < y_1, x_{\pi(2)} < y_2, ..., x_{\pi(d)} < y_d$.

a. Chứng tỏ hệ thức lồng ghép là bắc cầu.

b. Mô tả một phương pháp hiệu quả để xác định có hay không một hộp chiều-d lồng ghép bên trong một hộp khác.

c. Giả sử bạn có một tập hợp n hộp chiều-d $\{B_1, B_2,..., B_n\}$. Mô tả một thuật toán hiệu quả để xác định dãy dài nhất $<B_{i_1}, B_{i_2},..., B_{i_k}>$ gồm các hộp sao cho B_{i_j} lồng ghép trong $B_{i_{j+1}}$, for $j = 1, 2,..., k - 1$. Biểu diễn thời gian thực hiện của thuật toán theo dạng n và d.

25-3 Buôn chứng khoán

Buôn chứng khoán là việc vận dụng những điểm không thống nhất trong các tỷ giá hối đoái để biến đổi một đơn vị của một tiền tệ thành nhiều đơn vị của cùng tiền tệ đó. Ví dụ, giả sử 1 USD mua được 0.7 bảng Anh, 1 bảng Anh mua được 9.5 franc Pháp, và 1 franc Pháp mua được 0.16 USD. Như vậy, qua việc cập nhật các tiền tệ, một lái buôn có thể bắt đầu với 1 USD và mua 0.7 x 9.5 x 0.16 = 1.064 USD, như vậy chuyển một khoản lợi nhuận 6.4 phần trăm.

Giả sử ta có n tiền tệ $c_1, c_2,..., c_n$ và một bảng $n \times n$ R gồm các tỷ giá hối đoái, sao cho một đơn vị tiền tệ c_i mua được $R[i, j]$ đơn vị tiền tệ c_j.

a. Nêu một thuật toán hiệu quả để xác định ở đó có tồn tại một dãy các tiền tệ $<c_{i_1}, c_{i_2},..., c_{i_k}>$ hay không sao cho

$R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1$.

Phân tích thời gian thực hiện của thuật toán.

b. Nêu một thuật toán hiệu quả để in một dãy như vậy nếu như có. Phân tích thời gian thực hiện của thuật toán.

25-4 *Thuật toán định tỷ lệ của Gabow cho các lộ trình ngắn nhất nguồn đơn*

Thuật toán định tỷ lệ [scaling algorithm] giải quyết một bài toán bằng

cách trước tiên chỉ xét bit cấp cao nhất của mỗi giá trị nhập liệu có liên quan (như một trọng số cạnh). Sau đó, nó tu chính giải pháp ban đầu bằng cách xem xét hai bit cấp cao nhất. Nó dần dần xem xét ngày càng nhiều bit cấp cao hơn, mỗi lần cứ thế tu chính giải pháp, cho đến khi tất cả các bit đều được xét và giải pháp đúng đắn đã được tính toán.

Trong bài toán này, ta xét một thuật toán để tính toán các lộ trình ngắn nhất từ một nguồn đơn bằng cách định tỷ lệ các trọng số cạnh. Ta có một đồ thị có hướng $G = (V, E)$ với các trọng số cạnh không âm số nguyên w. Cho $W = \max_{(u,v)\in E}\{w(u,v)\}$. Mục tiêu của chúng ta đó là phát triển một thuật toán chạy trong $O(E \lg W)$ thời gian.

Thuật toán lần lượt khám phá từng bit một trong phần biểu diễn nhị phân của các trọng số cạnh, từ bit quan trọng nhất đến bit ít quan trọng nhất. Cụ thể, cho $k = \lceil \lg(W + 1) \rceil$ là số lượng các bit trong phần biểu diễn nhị phân của W, và cho $i = 1, 2,..., k$, cho $w_i(u, v) = \lfloor w(u, v)/2^{k-i} \rfloor$. Nghĩa là, $w_i(u, v)$ là phiên bản "giảm tỷ lệ" của $w(u, v)$ được cung cấp bởi i bit quan trọng nhất của $w(u, v)$. (Như vậy, $w_k(u, v) = w(u, v)$ với tất cả $(u, v) \in E$.) Ví dụ, nếu $k = 5$ và $w(u, v) = 25$, có phần biểu diễn nhị phân (11001), thì $w_3(u, v) = \langle 110 \rangle = 6$. Để có một ví dụ khác với $k = 5$, nếu $w(u, v) = \langle 00100 \rangle = 4$, thì $w_3(u, v) = \langle 001 \rangle = 1$. - Ta hãy định nghĩa $\delta_i(u, v)$ là trọng số lộ trình ngắn nhất từ đỉnh u đến đỉnh v dùng hàm trọng số w_i. Như vậy, $\delta_k(u, v) = \delta(u, v)$ với tất cả $u, v \in V$. Với một đỉnh nguồn đã cho s, trước tiên thuật toán định tỷ lệ tính toán các trọng số lộ trình ngắn nhất $\delta_2(s, v)$ với tất cả $v \in V$, sau đó tính toán $\delta_k(s, v)$ với tất cả $v \in V$, và vân vân, cho đến khi nó tính toán $\delta_k(s, v)$ với tất cả $v \in V$. Ta mặc nhận xuyên suốt rằng $|E| \geq |V| - 1$, và ta sẽ thấy rằng việc tính toán δ_i từ δ_{i-1} sẽ mất $O(E)$ thời gian, sao cho nguyên cả thuật toán mất $O(kE) = O(E \lg W)$ thời gian.

a. Giả sử với tất cả các đỉnh $v \in V$, ta có $\delta(s, v) \leq |E|$. Chứng tỏ ta có thể tính toán $\delta(s, v)$ với tất cả $v \in V$ trong $O(E)$ thời gian.

b. Chứng tỏ ta có thể tính toán $\delta_1(s, v)$ với tất cả $v \in V$ trong $O(E)$ thời gian. giờ đây, ta tập trung tính toán δ_i từ δ_{i-1}.

c. Chứng minh rằng với $i = 2, 3,..., k$, hoặc $w_i(u, v) = 2w_{i-1}(u, v)$ hoặc $w_i(u, v) = 2w_{i-1}(u, v) + 1$. Sau đó, chứng minh

$$2\delta_{i-1}(s, v) \leq \delta_i(s, v) \leq 2\delta_{i-1}(s, v) + |V| - 1$$

với tất cả $v \in V$.

d. Định nghĩa với $i = 2, 3,..., k$ và tất cả $(u, v) \in E$,

$$\hat{w}_i(u, v) = w_i(u, v) + 2\delta_{i-1}(s, u) - 2\delta_{i-1}(s, v).$$

Chứng minh với $i = 2, 3, \ldots, k$ và tất cả $u, v \in V$, giá trị "gia trọng lại" $\hat{w}_i(u, v)$ của cạnh (u, v) là một số nguyên không âm.

e. Giờ đây, định nghĩa $\hat{\delta}_i(s, v)$ là trọng số lộ trình ngắn nhất từ s đến v dùng hàm trọng số $\hat{w}_i$. Chứng minh rằng với $i = 2, 3, \ldots, k$ và tất cả $v \in V$,

$$\delta_i(s, v) = \hat{\delta}_i(s, v) + 2\delta_{i-1}(s, v)$$

và rằng $\hat{\delta}_i(s, v) \leq |E|$.

f. Nêu cách tính toán $\delta_i(s, v)$ từ $\delta_{i-1}(s, v)$ với tất cả $v \in V$ trong $O(E)$ thời gian, và kết luận rằng $\delta(s, v)$ có thể được tính toán với tất cả $v \in V$ trong $O(E \lg W)$ thời gian.

25-5 *Thuật toán chu trình trọng số trung bình cực tiểu của Karp*

Cho $G = (V, E)$ là một đồ thị có hướng với hàm trọng số $w : E \rightarrow \mathbf{R}$, và cho $n = |V|$. Ta định nghĩa ***trọng số trung bình*** của một chu trình $c = <e_1, e_2, \ldots, e_k>$ của các cạnh trong E là

$$\mu(c) = \frac{1}{n} \sum_{i=1}^{k} w(e_i) .$$

Cho $\mu^* = \min_c \mu(c)$, ở đó c nằm trong phạm vi tất cả các chu trình có hướng trong G. Một chu trình c mà $\mu(c) = \mu^*$ được gọi là một ***chu trình trọng số trung bình cực tiểu***. Bài toán này nghiên cứu một thuật toán hiệu quả để tính toán μ^*.

Không để mất tính tổng quát, ta mặc nhận mọi đỉnh $v \in V$ là khả đụng từ một đỉnh nguồn $s \in V$. Cho $\delta(s, v)$ là trọng số của một lộ trình ngắn nhất từ s đến v, và cho $\delta_k(s, v)$ là trọng số của một lộ trình ngắn nhất từ s đến v bao gồm chính xác k cạnh. Nếu không có lộ trình nào từ s đến v có *chính xác k* cạnh, thì $\delta_k(s, v) = \infty$.

a. Chứng tỏ nếu $\mu^* = 0$, thì G không chứa các chu trình trọng số âm và $\delta(s, v) = \min_{0 \leq k \leq n-1} \delta_k(s, v)$ với tất cả các đỉnh $v \in V$.

b. Chứng tỏ nếu $\mu^* = 0$, thì

$$\max_{0 \leq k \leq n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} \geq 0$$

với tất cả các đỉnh $v \in V$. (*Mách nước:* Dùng cả hai tính chất từ phần (a).)

c. Cho c là một chu trình trọng số 0, và cho u và v là hai đỉnh bất kỳ trên c. Giả sử trọng số của lộ trình từ u đến v dọc theo chu trình là x. Chứng minh $\delta(s, v) = \delta(s, u) + x$. (*Mách nước:* Trọng số của lộ trình từ v đến u dọc theo chu trình đến $-x$.)

d. Chứng tỏ nếu $\mu^* = 0$, thì ở đó tồn tại một đỉnh v trên chu trình trọng số trung bình cực tiểu sao cho

$$\max_{0 \le k \le n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} = 0 .$$

(*Mách nước*: Chứng tỏ một lộ trình ngắn nhất đến bất kỳ đỉnh nào trên chu trình trọng số trung bình cực tiểu có thể được mở rộng dọc theo chu trình để thực hiện một lộ trình ngắn nhất đến đỉnh kế tiếp trên chu trình.)

e. Chứng tỏ nếu $\mu^* = 0$, thì

$$\min_{v \in V} \max_{0 \le k \le n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} = 0$$

f. Chứng tỏ nếu ta cộng một hằng t vào trọng số của mỗi cạnh của G, thì μ^* được tăng lên t. Dùng điều này để chứng tỏ

$$\mu^* = \min_{v \in V} \max_{0 \le k \le n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} .$$

g. Nêu một thuật toán $O(VE)$ thời gian để tính μ^*.

Ghi chú Chương

Thuật toán Dijkstra [55] xuất hiện vào năm 1959, nhưng không đề cập gì đến hàng đợi ưu tiên. Thuật toán Bellman-Ford dựa trên các thuật toán riêng biệt của Bellman [22] và Ford [71]. Bellman mô tả quan hệ của các lộ trình ngắn nhất với các hạn chế sai phân. Lawler [132] mô tả thuật toán thời gian tuyến tính cho các lộ trình ngắn nhất trong một dag.

Khi các trọng số cạnh là các số nguyên tương đối nhỏ, ta có thể dùng các thuật toán hiệu quả hơn để giải quyết bài toán các lộ trình ngắn nhất nguồnđơn. Ahuja, Mehlhorn, Orlin, và Tarjan [6] đưa ra một thuật toán chạy trong $O(E + \sqrt{\lg W})$ thời gian trên đồ thị có các trọng số cạnh không âm, ở đó W là trọng số lớn nhất của bất kỳ cạnh nào trong đồ thị. Họ cũng cung cấp một thuật toán dễ lập trình chạy trong $O(E + V \lg W)$ thời gian. Với đồ thị có các trọng số cạnh âm, thuật toán của Gabow và Tarjan [77] chạy trong $O(\sqrt{V}\ E\ \lg(\ V W))$ thời gian, ở đó W là độ lớn của trọng số độ lớn lớn nhất của bất kỳ cạnh nào trong đồ thị.

Papadimitriou và Steiglitz [154] đã có một tài liệu hay đề cập đến phương pháp đơn hình và thuật toán ellipsoid cũng như các thuật toán khác liên quan đến lập trình tuyến tính. Thuật toán đơn hình của lập

trình tuyến tính được G. Danzig sáng chế vào năm 1947. Các biến thức của đơn hình giữ lại các phương pháp phổ dụng nhất để giải quyết các bài toán lập trình tuyến tính. Thuật toán ellipsoid được xem là do L. G. Khachian sáng chế vào năm 1979, dựa trên công trình trước đó của N. Z. Shor, D. B. Judin, và A. S. Nemirovskii. Karmarkar mô tả thuật toán của ông trong [115].

26 Các Lộ Trình Ngắn Nhất Mọi Cặp

Trong chương này, ta xét bài toán tìm các lộ trình ngắn nhất giữa tất cả các cặp đỉnh trong một đồ thị. Bài toán này có thể nảy sinh trong khi tạo một bảng các khoảng cách giữa tất cả các cặp thành phố dùng cho một tập bản đồ đường. Cũng như trong Chương 25, ta có một đồ thị có hướng gia trọng $G = (V, E)$ với một hàm trọng số $w : E \rightarrow \mathbf{R}$ ánh xạ các cạnh theo các trọng số có giá trị thực. Ta muốn tìm ra, với mọi cặp đỉnh $u, v \in V$, một lộ trình ngắn nhất (trọng số ít nhất) từ u đến v, ở đó trọng số của một lộ trình là tổng các trọng số của các cạnh cấu thành của nó. Ta thường muốn kết xuất theo dạng bảng biểu: khoản nhập trong hàng của u và cột của v sẽ là trọng số của một lộ trình ngắn nhất từ u đến v.

Để giải quyết bài toán các lộ trình ngắn nhất mọi cặp, ta chạy một thuật toán các lộ trình ngắn nhất nguồn đơn $|V|$ lần, mỗi lần cho một đỉnh làm nguồn. Nếu tất cả các trọng số cạnh đều không âm, ta có thể dùng thuật toán Dijkstra. Nếu dùng cách thực thi mảng tuyến tính của hàng đợi ưu tiên, thời gian thực hiện là $O(V^3 + VE) = O(V^3)$. Cách thực thi đống nhị phân của hàng đợi ưu tiên sẽ cho ra một thời gian thực hiện là $O(VE \lg V)$, là một cải thiện nếu đồ thị thưa. Một cách khác, ta có thể thực thi hàng đợi ưu tiên với một đống Fibonacci, mang lại một thời gian thực hiện là $O(V^2 \lg V + VE)$.

Nếu các trọng số cạnh âm được phép, có thể ta không còn dùng được thuật toán Dijkstra. Thay vì thế, ta phải chạy thuật toán Bellman-Ford chậm hơn mỗi lần từ một đỉnh. Thời gian thực hiện kết quả là $O(V^2E)$, mà trên một đồ thị trù mật nó là $O(V^4)$. Chương này sẽ giải thích cách thực hiện tốt hơn. Ta sẽ cũng nghiên cứu quan hệ của bài toán các lộ trình ngắn nhất mọi cặp với phép nhân ma trận và nghiên cứu cấu trúc đại số của nó.

Khác với các thuật toán nguồnđơn, mặc nhận một phép biểu diễn danh sách kề của đồ thị, hầu hết thuật toán trong chương này đều dùng một phép biểu diễn ma trận kề. (Thuật toán Johnson cho đồ thị thưa sử dụng các danh sách kề.) Nhập liệu là một ma trận $n \times n$ W biểu thị cho các trọng số cạnh của một đồ thị có hướng đỉnh -n $G = (V, E)$. Nghĩa là, $W = (w_{ij})$, ở đó

$$w_{ij} = \begin{cases} 0 & \text{nếu } i = j, \\ \text{trọng số của cạnh có hướng } (i, j) & \text{nếu } i \neq j \text{ và } (i, j) \in E, \\ \infty & \text{nếu } i \neq j \text{ và } (i, j) \notin E. \end{cases} \quad (26.1)$$

Các trọng số cạnh âm được phép, nhưng trước mắt ta mặc nhận đồ thị nhập liệu không chứa các chu trình trọng số âm.

Kết xuất bảng của các thuật toán các lộ trình ngắn nhất mọi cặp được trình bày trong chương này là một ma trận $n \times n$ $D = (d_{ij})$, ở đó khoản nhập d_{ij} chứa trọng số của một lộ trình ngắn nhất từ đỉnh i đến đỉnh j. Nghĩa là, nếu ta cho $\delta(i, j)$ thể hiện trọng số lộ trình ngắn nhất từ đỉnh i đến đỉnh j (như trong Chương 25), thì $d_{ij} = \delta(i, j)$ vào lúc kết thúc.

Để giải quyết bài toán các lộ trình ngắn nhất mọi cặp trên một nhập liệu ma trận kề, ta cần tính toán không những các trọng số lộ trình ngắn nhất mà còn một ***ma trận phần tử tiền vị*** $\Pi = (\pi_{ij})$, ở đó π_{ij} là NIL nếu $i = j$ hoặc không có lộ trình bào từ i đến j, và bằng không π_{ij} là một phần tử tiền vị nào đó của j trên một lộ trình ngắn nhất từ i. Cũng như đồ thị con phần tử tiền vị G_π trong Chương 25 là một cây các lộ trình ngắn nhất đối với một đỉnh nguồn đã cho, đồ thị con được cảm sinh bởi hàng thứ i của ma trận Π phải là một cây các lộ trình ngắn nhất có gốc i. Với mỗi đỉnh $i \in V$, ta định nghĩa ***đồ thị con phần tử tiền vị*** của G với i là $G_{\pi,i} = (V_{\pi,i}, E_{\pi,i})$, ở đó

$$V_{\pi,i} = \{ j \in V : \pi_{ij} \neq \text{NIL}\} \cup \{i\}$$

và

$$E_{\pi,i} = \{(\pi_{ij}, j) : j \in V_{\pi,i} \text{ và } \pi_{ij} \neq \text{NIL}\}$$

Nếu $G_{\pi,i}$ là một cây các lộ trình ngắn nhất, thì thủ tục dưới đây, là một phiên bản đã sửa đổi của thủ tục PRINT-PATH trong Chương 23, sẽ in một lộ trình ngắn nhất từ đỉnh i đến đỉnh j.

```
PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, j)

1  if  i = j
2      then print i
3      else if π_ij = NIL
4              then print "không có lộ trình nào từ " i "đến" j "tồn tại"
5              else PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, π_ij)
6                   print j
```

Để nêu bật các tính năng thiết yếu của các thuật toán mọi cặp trong chương này, ta sẽ không đề cập cách tạo và các tính chất của các ma

trận phần tử tiền vị nhiều như trường hợp các đồ thị con phần tử tiền vị trong Chương 25. Những điểm cơ bản sẽ được một số bài tập đề cập.

Khái quát chương

Đoạn 26.1 trình bày một thuật toán lập trình động dựa trên phép nhân ma trận để giải quyết bài toán các lộ trình ngắn nhất mọi cặp. Dùng kỹ thuật "bình phương lặp lại," có thể tạo thuật toán này để chạy trong $\Theta(V^3 \lg V)$ thời gian. Đoạn 26.2 sẽ đề cập một thuật toán lập trình động khác, thuật toán Floyd-Warshall. Thuật toán Floyd-Warshall chạy trong thời gian $\Theta(V^3)$. Đoạn 26.2 cũng đề cập bài toán tìm bao đóng bắc cầu [transitive closure] của một đồ thị có hướng, liên quan đến bài toán các lộ trình ngắn nhất mọi cặp. Thuật toán Johnson được trình bày trong Đoạn 26.3. Khác với các thuật toán khác trong chương này, thuật toán Johnson sử dụng phép biểu diễn danh sách kề của một đồ thị. Nó giải quyết bài toán các lộ trình ngắn nhất mọi cặp trong $O(V^2 \lg V + VE)$ thời gian, khiến nó trở thành một thuật toán hay cho đồ thị thưa, lớn. Cuối cùng, trong Đoạn 26.4, ta xét một cấu trúc đại số tên "nửa vành đóng" [closed semiring], cho phép áp dụng nhiều thuật toán các lộ trình ngắn nhất cho một loạt các bài toán mọi cặp khác không bao hàm các lộ trình ngắn nhất.

Trước khi tiến hành, ta cần thiết lập vài quy ước cho các phép biểu diễn ma trận kề. Trước tiên, ta sẽ mặc nhận chung rằng đồ thị nhập liệu $G = (V, E)$ có n đỉnh, sao cho $n = |V|$. Thứ hai, ta sẽ dùng quy ước thể hiện các ma trận bằng các mẫu tự hoa, như W hoặc D, và các thành phần riêng lẻ của chúng bằng các con chữ dưới viết thường, như w_{ij} hoặc d_{ij}. Một số ma trận sẽ có các con chữ trên đóng ngoặc đơn, như trong $D^{(m)} = (d_{ij}^{(m)})$, để nêu rõ các lần lặp lại. Cuối cùng, với một ma trận đã cho $n \times n$ A, ta sẽ mặc nhận rằng giá trị của n được lưu trữ trong thuộc tính *rows*[*A*].

26.1 Các lộ trình ngắn nhất và phép nhân ma trận

Đoạn này trình bày một thuật toán lập trình động cho bài toán các lộ trình ngắn nhất mọi cặp trên một đồ thị có hướng $G = (V, E)$. Mỗi vòng lặp chính của chương trình động sẽ triệu gọi một phép toán tương tự như phép nhân ma trận, sao cho thuật toán trông giống như phép nhân ma trận lặp lại. Ta sẽ bắt đầu bằng việc phát triển một thuật toán $\Theta(V^4)$ thời gian cho bài toán các lộ trình ngắn nhất mọi cặp rồi cải thiện thời gian thực hiện củanó theo $\Theta(V^3 \lg V)$.

Trước khi tiến hành, ta hãy sơ lược các bước nêu trong Chương 16 về

tiến trình phát triển một thuật toán lập trình động.

1. Định rõ đặc điểm cấu trúc của một giải pháp tối ưu.

2. Định nghĩa giá trị của một giải pháp tối ưu theo đệ quy.

3. Tính toán giá trị của một giải pháp tối ưu theo dạng dưới lên.

(Bước thứ tư, kiến tạo một giải pháp tối ưu từ thông tin đã tính toán, sẽ dành cho các bài tập.)

Cấu trúc của lộ trình ngắn nhất

Để bắt đầu, ta định rõ đặc điểm cấu trúc của một giải pháp tối ưu. Với bài toán các lộ trình ngắn nhất mọi cặp trên một đồ thị $G = (V, E)$, ta đã chứng minh (Bổ đề 25.1) rằng tất cả các lộ trình con của một lộ trình ngắn nhất chính là các lộ trình ngắn nhất. Giả sử rằng đồ thị được biểu thị bởi một ma trận kề $W = (w_{ij})$. Xét một lộ trình ngắn nhất p từ đỉnh i đến đỉnh j, và giả sử p chứa tối đa m cạnh. Giả sử ta không có các chu trình trọng số âm, m là hữu hạn. Nếu $i = j$, thì p có trọng số 0 và không có các cạnh. Nếu các đỉnh i và j riêng biệt, ta phân tích lộ trình p thành $i \overset{p'}{\leadsto} k \rightarrow j$, ở đó lộ trình p' giờ đây chứa tối đa $m - 1$ cạnh. Hơn nữa, theo Bổ đề 25.1, p' là một lộ trình ngắn nhất từ i đến k. Như vậy, theo Hệ luận 25.2, ta có $\delta(i, j) = \delta(i, k) + w_{kj}$.

Một giải pháp đệ quy cho bài toán các lộ trình ngắn nhất mọi cặp

Giờ đây, cho $d_{ij}^{(m)}$ là trọng số cực tiểu của bất kỳ lộ trình nào từ đỉnh i đến đỉnh j chứa tối đa m cạnh. Khi $m = 0$, ta có một lộ trình ngắn nhất từ i đến j mà không có các cạnh nếu và chỉ nếu $i = j$. Như vậy,

$$d_{ij}^{(0)} = \begin{cases} 0 & \text{nếu } i = j, \\ \infty & \text{nếu } i \neq j. \end{cases}$$

Với $m \geq 1$, ta tính toán $d_{ij}^{(m)}$ dưới dạng cực tiểu của $d_{ij}^{(m-1)}$ (trọng số của lộ trình ngắn nhất từ i đến j bao gồm tối đa m - 1 cạnh) và trọng số cực tiểu của bất kỳ lộ trình từ i đến j bao gồm tối đa m cạnh, có được nhờ xem xét tất cả các phần tử tiền vị khả dĩ k của j. Như vậy, ta định nghĩa theo đệ quy

$$\begin{aligned} d_{ij}^{(m)} &= \min\left(d_{ij}^{(m-1)}, \min_{1 \leq k \leq n}\left\{d_{ik}^{(m-1)} + w_{kj}\right\}\right) \\ &= \min_{1 \leq k \leq n}\left\{d_{ij}^{(m-1)} + w_{kj}\right\}. \end{aligned} \qquad (26.2)$$

Đẳng thức sau là đúng bởi $w_{jj} = 0$ với tất cả j.

Đâu là các trọng số lộ trình ngắn nhất thực tế $\delta(i, j)$? Nếu đồ thị không chứa các chu trình trọng số âm, thì tất cả các lộ trình ngắn nhất là đơn giản và như vậy chứa tối đa n - 1 cạnh. Một lộ trình từ đỉnh i đến

đỉnh j với trên n - 1 cạnh không thể có ít trọng số hơn một lộ trình ngắn nhất từ i đến j. Do đó, các trọng số lộ trình ngắn nhất thực tế là từ

$$\delta(i,j) = d_{ij}^{(n-1)} = d_{ij}^{(n)} = d_{ij}^{(n+1)} = \ldots \quad . \qquad (26.3)$$

Tính toán các trọng số lộ trình ngắn nhất dưới lên

Để nhập liệu, ta lấy ma trận $W = (w_{ij})$, giờ đây tính toán một sêri các ma trận $D^{(1)}, D^{(2)}, \ldots, D^{(n-1)}$, ở đó với $m = 1, 2, \ldots, n - 1$, ta có $D^{(m)} = (d_{ij}^{(m)})$. Ma trận chung cuộc $D^{(n-1)}$ chứa các trọng số lộ trình ngắn nhất thực tế. Nhận thấy do $d_{ij}^{(1)} = w_{ij}$ với tất cả các đỉnh $i, j \in V$, nên ta có $D^{(1)} = W$.

Trọng tâm của thuật toán là thủ tục dưới đây, mà, căn cứ vào các ma trận $D^{(m-1)}$ và W, nó trả về ma trận $D^{(m)}$. Nghĩa là, nó mở rộng thêm một cạnh cho các lộ trình ngắn nhất đã tính toán cho đến giờ.

```
EXTEND-SHORTEST-PATHS(D, W)
n ← rows[D]
cho D' = (d'_ij) là một ma trận n x n
for i ← 1 to n
     do for j ← 1 to n
            do d'_ij ← ∞
               for k ← 1 to n
                   do d'_ij - min(d'_ij, d_ik + w_kj)
return D'
```

Thủ tục tính toán một ma trận $D' = (d'_{ij})$, mà nó trả về vào cuối. Để thực hiện, nó tính toán phương trình (26.2) với tất cả i và j, dùng D với $D^{(m-1)}$ và D' với $D^{(m)}$. (Nó được viết mà không có các con chữ trên để các ma trận nhập liệu và kết xuất củanó độc lập với m.) Thời gian thực hiện của nó là $\Theta(n^3)$ do ba vòng lặp **for** được lồng.

Giờ đây, ta có thể thấy quan hệ với phép nhân ma trận. Giả sử ta muốn tính tích ma trận $C = A \cdot B$ của hai ma trận n x n A và B. Như vậy, với $i, j = 1, 2, \ldots, n$, ta tính toán

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \, . \qquad (26.4)$$

Nhận thấy nếu ta tiến hành các phép thay

$$d^{(m-1)} \rightarrow a \, ,$$
$$w \rightarrow b \, ,$$
$$d^{(m)} \rightarrow c \, ,$$
$$\min \rightarrow + \, ,$$

$+ \rightarrow \cdot$

trong phương trình (26.2), ta được phương trình (26.4). Như vậy, nếu thực hiện các thay đổi này đối với EXTEND-SHORTEST-PATHS và cũng thay ∞ (đồng nhất thức của min) bằng 0 (đồng nhất thức của +), ta được thủ tục đơn giản $\Theta(n^3)$ - thời gian cho phép nhân ma trận.

```
MATRIX-MULTIPLY(A, B)
n ← rows[A]
cho C là một ma trận n X n
for i ← 1 to n
     do for j ← 1 to n
            do c_ij ← 0
               for k ← 1 to n
                   do c_ij ← c_ij + a_ik • b_kj
return C
```

Trở về bài toán các lộ trình ngắn nhất mọi cặp, ta tính toán các trọng số lộ trình ngắn nhất bằng cách mở rộng các lộ trình ngắn nhất theo từng cạnh. Việc cho $A \cdot B$ thể hiện "tích" ma trận do EXTEND-SHORTEST-PATHS(A, B) trả về, ta tính toán dãy n - 1 ma trận

$$
\begin{aligned}
D^{(1)} &= D^{(0)} \cdot W = W, \\
D^{(2)} &= D^{(1)} \cdot W = W^2, \\
D^{(3)} &= D^{(2)} \cdot W = W^3, \\
&\vdots \\
D^{(n-1)} &= D^{(n-2)} \cdot W = W^{n-1}.
\end{aligned}
$$

Như ta đã chứng tỏ trên đây, ma trận $D^{(n-1)} = W^{n-1}$ chứa các trọng số lộ trình ngắn nhất. Thủ tục dưới đây tính toán dãy này trong $\Theta(n^4)$ thời gian.

```
SLOW-ALL-PAIRS-SHORTEST-PATHS(W)
n ← rows[W]
D^(1) ← W
for m ← 2 to n - 1
    do D^(m) ← EXTEND-SHORTEST-PATHS(D^(m-1), W)
return D^(n-1)
```

Hình 26.1 nêu một đồ thị và các ma trận $D(m)$ đã tính toán bởi thủ tục SLOW-ALL-PAIRS-SHORTEST-PATHS.

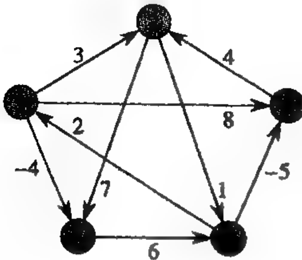
Cải thiện thời gian thực hiện

Tuy nhiên, mục tiêu của chúng ta là không tính toán *tất cả* các ma trận $D^{(m)}$: ta chỉ quan tâm đến ma trận $D^{(n-1)}$, Hãy nhớ lại nếu không có các chu trình trọng số âm, phương trình (26.3) hàm ý $D^{(m)} = D^{(n-1)}$ với tất cả các số nguyên $m \geq n-1$. Ta có thể tính toán $D^{(n-1)}$ với chỉ $\lceil \lg(n-1) \rceil$ tích ma trận bằng cách tính toán dãy

$$
\begin{aligned}
D^{(1)} &= W, \\
D^{(2)} &= W^2 = W \cdot W, \\
D^{(4)} &= W^4 = W^2 \cdot W^2 \\
D^{(8)} &= W^8 = W^4 \cdot W^4, \\
&\vdots \\
D^{(2^{\lceil \lg(n-1) \rceil})} &= W^{2^{\lceil \lg(n-1) \rceil}} = W^{2^{\lceil \lg(n-1) \rceil - 1}} \cdot W^{2^{\lceil \lg(n-1) \rceil - 1}}.
\end{aligned}
$$

Bởi $2^{\lceil \lg(n-1) \rceil} \geq n-1$, tích chung cuộc $D^{(2^{\lceil \lg(n-1) \rceil})}$ sẽ bằng với $D^{(n-1)}$.

Thủ tục dưới đây tính toán dãy các ma trận trên đây bằng cách dùng kỹ thuật ***bình phương lặp lại***.



$$
D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}
\quad
D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}
$$

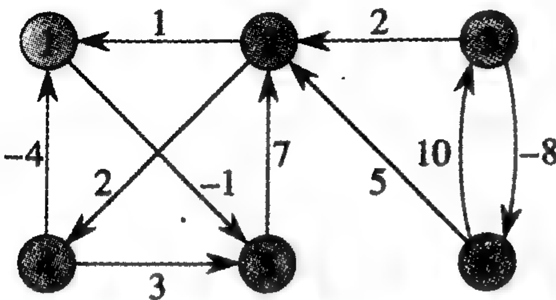
$$
D^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}
\quad
D^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}
$$

Hình 26.1 Một đồ thị có hướng và dãy các ma trận $D^{(m)}$ được SLOW-ALL-PAIRS-SHORTEST-PATHS tính toán. Độc giả có thể xác minh rằng $D^{(5)} = D^{(4)}$. W bằng với $D^{(4)}$, và như vậy $D^{(m)} = D^{(4)}$ với tất cả $m \geq 4$.

```
FASTER-ALL-PAIRS-SHORTEST-PATHS(W)
1 n ← rows[W]
2 D^(1) ← W
3 m ← 1
4 while n − 1 > m
5         do D^(2m) ← EXTEND-SHORTEST-PATHS(D^(m), D^(m))
6            m ← 2m
7 return D^(m)
```

Trong mỗi lần lặp lại của vòng lặp **while** trong các dòng 4-6, ta tính toán $D^{(2m)} = (D^{(m)})^2$, bắt đầu với $m = 1$. Vào cuối mỗi lần lặp lại, ta nhân đôi giá trị của m. Lần lặp lại cuối sẽ tính toán $D^{(n-1)}$ bằng cách thực tế tính toán $D^{(2m)}$ với một $n - 1 \leq 2m < 2n - 2$. Theo phương trình (26.3), $D^{(2m)} = D^{(n-1)}$. Vào lần sau, đợt kiểm tra trong dòng 4 được thực hiện, m đã được nhân đôi, do đó giờ đây $n - 1 \leq m$, đợt kiểm tra thất bại, và thủ tục trả về ma trận cuối mà nó đã tính toán.

Thời gian thực hiện của FASTER-ALL-PAIRS-SHORTEST-PATHS là $\Theta(n^3 \lg n)$ bởi mỗi trong số $\lceil \lg(n - 1) \rceil$ tích ma trận mất $\Theta(n^3)$ thời gian. Nhận thấy mã là chặt, không chứa các cấu trúc dữ liệu tinh vi, và do đó hằng ẩn trong hệ ký hiệu Θ là nhỏ.



Hình 26.2 Một đồ thị có hướng gia trọng để dùng trong Bài tập 26.1-1, 26.2-1, và 26.3-1.

Bài tập

26.1-1

Chạy SLOW-ALL-PAIRS-SHORTEST-PATHS trên đồ thị có hướng gia trọng của Hình 26.2, nêu các ma trận kết quả của mỗi lần lặp lại của các vòng lặp tương ứng. Sau đó, thực hiện y như vậy với FASTER-ALL-PAIRS-SHORTEST-PATHS.

26.1-2

Tại sao ta lại yêu cầu $w_{ii} = 0$ với tất cả $1 \le i \le n$?

26.1-3

Ma trận

$$D^{(0)} = \begin{pmatrix} 0 & \infty & \infty & \cdots & \infty \\ \infty & 0 & \infty & \cdots & \infty \\ \infty & \infty & 0 & \cdots & \infty \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \infty & \infty & \infty & \cdots & 0 \end{pmatrix}$$

được dùng trong các thuật toán các lộ trình ngắn nhất tương ứng với gì trong phép nhân ma trận bình thường?

26.1-4

Nêu cách diễn tả bài toán các lộ trình ngắn nhất nguồnđơn dưới dạng một tích các ma trận và một vectơ. Mô tả cách đánh giá tích này tương ứng với một thuật toán tương tự Bellman-Ford (xem Đoạn 25.3).

26.1-5

Giả sử ta cũng muốn tính toán các đỉnh trên các lộ trình ngắn nhất trong các thuật toán của đoạn này. Nêu cách tính toán ma trận phần tử tiền vị Π từ ma trận hoàn thành D của các trọng số lộ trình ngắn nhất trong $O(n^3)$ thời gian.

26.1-6

Các đỉnh trên các lộ trình ngắn nhất cũng có thể được tính toán cùng lúc với các trọng số lộ trình ngắn nhất. Ta hãy định nghĩa $\pi_{ij}^{(m)}$ là phần tử tiền vị của đỉnh j trên bất kỳ lộ trình trọng số cực tiểu nào từ i đến j chứa tối đa m cạnh. Sửa đổi EXTEND-SHORTEST-PATHS và SLOW-ALL-PAIRS-SHORTEST-PATHS để tính toán các ma trận $\Pi^{(1)}, \Pi^{(2)}, ..., \Pi^{(n-1)}$ khi các ma trận $D^{(1)}, D^{(2)},..., D^{(n-1)}$ được tính toán.

26.1-7

Thủ tục FASTER-ALL-PAIRS-SHORTEST-PATHS, như được viết, yêu cầu ta lưu trữ $\lceil \lg(n-1) \rceil$ ma trận, mỗi ma trận có n^2 thành phần, với yêu cầu tổng không gian là $\Theta(n^2 \lg n)$. Sửa đổi thủ tục để chỉ yêu cầu $\Theta(n^2)$ không gian bằng cách chỉ dùng hai ma trận $n \times n$.

26.1-8

Sửa đổi FASTER-ALL-PAIRS-SHORTEST-PATHS để phát hiện sự

hiện diện của một chu trình trọng số âm.

26.1-9

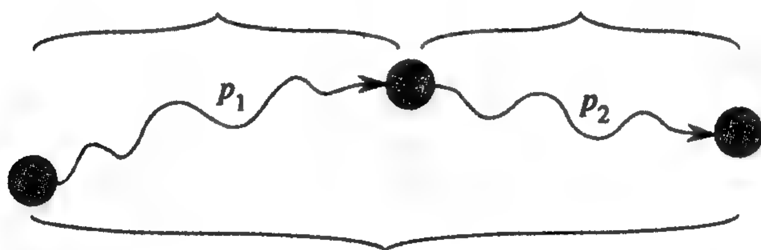
Nêu một thuật toán hiệu quả để tìm chiều dài (số cạnh) của một chu trình trọng số âm có chiều dài cực tiểu trong một đồ thị.

26.2 Thuật toán Floyd-Warshall

Trong đoạn này, ta sẽ dùng một cách trình bày lập trình động khác để giải quyết bài toán các lộ trình ngắn nhất mọi cặp trên một đồ thị có hướng $G = (V, E)$. Thuật toán kết quả, có tên là ***thuật toán Floyd-Warshall***, chạy trong $\Theta(V^3)$ thời gian. Như trên đây, các trọng số cạnh âm có thể hiện diện, nhưng ta mặc nhận rằng không có các chu trình trọng số âm. Cũng như trong Đoạn 26.1, ta sẽ theo tiến trình lập trình động để phát triển thuật toán. Sau khi nghiên cứu thuật toán kết quả, ta sẽ trình bày một phương pháp tương tự để tìm bao đóng bắc cầu [transitive closure] của một đồ thị có hướng.

Cấu trúc của một lộ trình ngắn nhất

Trong thuật toán Floyd-Warshall, ta dùng một kiểu mô tả đặc điểm khác về cấu trúc của một lộ trình ngắn nhất so với kiểu đã dùng trong các thuật toán mọi cặp gốc phép nhân ma trận. Thuật toán xét các đỉnh "trung gian" của một lộ trình ngắn nhất, ở đó một ***đỉnh trung gian*** của một lộ trình đơn giản $p = \langle v_1, v_2, \ldots, v_l \rangle$ là bất kỳ đỉnh nào của p khác ngoài v_1 hoặc v_l, nghĩa là, bất kỳ đỉnh nào trong tập hợp $\{v_2, v_3, \ldots, v_{l-1}\}$.



Hình 26.3 Lộ trình p là một lộ trình ngắn nhất từ đỉnh i đến đỉnh j, và k là đỉnh trung gian được đánh số cao nhất của p. Lộ trình p_1, phần lộ trình p từ đỉnh i đến đỉnh k, có tất cả các đỉnh trung gian trong tập hợp $\{1, 2, \ldots, k-1\}$. Điều này cùng áp dụng cho lộ trình p_2 từ đỉnh k đến đỉnh j.

Thuật toán Floyd-Warshall dựa trên nhận xét dưới đây. Cho các đỉnh của G là $V = \{1, 2,..., n\}$, và xét một tập hợp con $\{1, 2,..., k\}$ các đỉnh với vài k. Với một cặp đỉnh i, $j \in V$ bất kỳ, xét tất cả các lộ trình từ i đến j có tất cả các đỉnh trung gian được rút từ $\{1, 2,..., k\}$, và cho p là một lộ trình trọng số cực tiểu từ trong số chúng. (Lộ trình p là đơn giản, bởi ta mặc nhận G không chứa các chu trình trọng số âm.) Thuật toán Floyd-Warshall khai thác một mối quan hệ giữa lộ trình p và các lộ trình ngắn nhất từ i đến j với tất cả các đỉnh trung gian trong tập hợp $\{1, 2,..., k - 1\}$. Mối quan hệ tùy thuộc vào việc k có phải là một đỉnh trung gian của lộ trình p hay không.

- Nếu k không phải là một đỉnh trung gian của lộ trình p, thì tất cả các đỉnh trung gian của lộ trình p nằm trong tập hợp $\{1, 2,..., k - 1\}$. Như vậy, một lộ trình ngắn nhất từ đỉnh i đến đỉnh j với tất cả các đỉnh trung gian trong tập hợp $\{1, 2,..., k - 1\}$ cũng là một lộ trình ngắn nhất từ i đến j với tất cả các đỉnh trung gian trong tập hợp $\{1,2,...,k\}$.

- Nếu k là một đỉnh trung gian của lộ trình p, thì ta tách nhỏ p thành $i \overset{p_1}{\leadsto} k \overset{p_2}{\leadsto} j$ như đã nêu trong Hình 26.3. Theo Bổ đề 25.1, p_1 là một lộ trình ngắn nhất từ i đến k với tất cả các đỉnh trung gian trong tập hợp $\{1, 2,..., k\}$. Thực vậy, đỉnh k không phải là một đỉnh trung gian của lộ trình p_1, và do đó p_1 là một lộ trình ngắn nhất từ i đến k với tất cả các đỉnh trung gian trong tập hợp $\{1, 2,..., k - 1\}$. Cũng vậy, p_2 là một lộ trình ngắn nhất từ đỉnh k đến đỉnh j với tất cả các đỉnh trung gian trong tập hợp $\{1, 2,..., k - 1\}$.

Một giải pháp đệ quy cho bài toán các lộ trình ngắn nhất mọi cặp

Dựa vào các nhận xét trên đây, ta định nghĩa một cách trình bày đệ quy khác về các ước lượng lộ trình ngắn nhất so với trong Đoạn 26.1. Cho $d_{ij}^{(k)}$ là trọng số của một lộ trình ngắn nhất từ đỉnh i đến đỉnh j với tất cả các đỉnh trung gian trong tập hợp $\{1, 2,..., k\}$. Khi $k = 0$, một lộ trình từ đỉnh i đến đỉnh j mà không có đỉnh trung gian nào được đánh số cao hơn 0 sẽ không có các đỉnh trung gian nào cả. Như vậy nó có tối đa một cạnh, và do đó $d_{ij}^{(0)} = w_{ij}$. Một định nghĩa đệ quy dựa vào

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{nếu } k = 0, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{nếu } k \geq 1. \end{cases} \qquad (26.5)$$

Ma trận $D^{(n)} = \left(d_{ij}^{(n)}\right)$ cho đáp án chung cuộc $d_{ij}^{(n)} = \delta(i, j)$ với tất cả $i, j \in V$—bởi tất cả các đỉnh trung gian nằm trong tập hợp $\{1, 2,..., n\}$.

Tính toán các trọng số lộ trình ngắn nhất dưới lên

Dựa trên phép truy toán (26.5), ta có thể dùng thủ tục dưới lên sau

đây để tính toán các giá trị $d_{ij}^{(k)}$ theo thứ tự của các giá trị tăng dần của k. Nhập liệu của nó là một ma trận $n \times n$ W được định nghĩa như trong phương trình (26.1). Thủ tục trả về ma trận $D^{(n)}$ gồm các trọng số lộ trình ngắn nhất.

```
FLOYD-WARSHALL(W)
1  n ← rows[W]
2  D^(0) ← W
3  for k ← 1 to n
4       do for i ← 1 to n
5              do for j ← 1 to n
6                     d_ij^(k) ← min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
7  return D^(n)
```

Hình 26.4 nêu một đồ thị có hướng và các ma trận $D^{(k)}$ đã tính toán bằng thuật toán Floyd-Warshall.

Thời gian thực hiện của thuật toán Floyd-Warshall được xác định bằng các vòng lặp **for** được lồng ba trong các dòng 3-6. Mỗi lần thi hành dòng 6 mất $O(1)$ thời gian. Do đó, thuật toán chạy trong thời gian $\Theta(n^3)$. Như trong thuật toán chung cuộc trong Đoạn 26.1, mã là chặt, không có các cấu trúc dữ liệu tinh vi, và do đó hằng ẩn trong hệ ký hiệu Θ là nhỏ. Như vậy, thuật toán Floyd-Warshall khá thực tiễn với cả các đồ thị nhập liệu có quy mô vừa phải.

Kiến tạo một lộ trình ngắn nhất

Có nhiều phương pháp khác nhau để kiến tạo các lộ trình ngắn nhất trong thuật toán Floyd-Warshall. Một cách đó là tính toán ma trận D của các trọng số lộ trình ngắn nhất rồi kiến tạo ma trận phần tử tiền vị Π từ ma trận D. Phương pháp này có thể được thực thi để chạy trong $O(n^3)$ thời gian (Bài tập 26.1-5). Cho ma trận phần tử tiền vị Π, thủ tục PRINT-ALL-PAIRS-SHORTEST-PATH có thể được dùng để in các đỉnh trên một lộ trình ngắn nhất đã cho.

Ta có thể tính toán ma trận phần tử tiền vị Π "trực tuyến" hệt như thuật toán Floyd-Warshall tính toán các ma trận $D^{(k)}$. Cụ thể, ta tính toán một dãy các ma trận $\Pi^{(0)}$, $\Pi^{(1)}$, ..., $\Pi^{(n)}$, ở đó $\Pi = \Pi^{(n)}$ và $\pi_{ij}^{(k)}$ được định nghĩa là phần tử tiền vị của đỉnh j trên một lộ trình ngắn nhất từ đỉnh i với tất cả các đỉnh trung gian trong tập hợp $\{1, 2, ..., k\}$.

Ta có thể đưa ra một cách trình bày đệ quy của $\pi_{ij}^{(k)}$. Khi $k = 0$, một lộ trình ngắn nhất từ i đến j không có các đỉnh trung gian nào cả. Do đó,

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Hình 26.4 Dãy các ma trận $D^{(k)}$ và $\Pi^{(k)}$ được tính toán bởi thuật toán Floyd-Warshall với đồ thị trong Hình 26.1.

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{nếu } i = j \text{ hoặc } w_{ij} = \infty, \\ i & \text{nếu } i \neq j \text{ và } w_{ij} < \infty. \end{cases} \quad (26.6)$$

Với $k \geq 1$, nếu ta lấy lộ trình $i \rightsquigarrow k \rightsquigarrow j$, thì phần tử tiền vị của j mà ta chọn sẽ giống như phần tử tiền vị của j mà ta đã chọn trên một lộ trình ngắn nhất từ k với tất cả các đỉnh trung gian trong tập hợp $\{1, 2,..., k - 1\}$. Bằng không, ta cũng chọn phần tử tiền vị của j mà ta đã chọn trên một lộ trình ngắn nhất từ i với tất cả các đỉnh trung gian trong tập hợp $\{1, 2,..., k - 1\}$. Chính thức, với $k \geq 1$,

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{nếu } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{nếu } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases} \quad (26.7)$$

Ta để việc sát nhập các phép tính ma trận $\Pi^{(k)}$ vào thủ tục FLOYD-WARSHALL làm Bài tập 26.2-3. Hình 26.4 có nêu dãy các ma trận $\Pi^{(k)}$ mà thuật toán kết quả tính toán cho đồ thị của Hình 26.1. Bài tập cũng yêu cầu khó hơn khi phải chứng minh rằng đồ thị con phần tử tiền vị $G_{\pi,i}$ là một cây các lộ trình ngắn nhất có gốc i. Bài tập 26.2-6 cũng nêu ra một cách khác để kiến tạo lại các lộ trình ngắn nhất.

Bao đóng bắc cầu của một đồ thị có hướng

Cho một đồ thị có hướng $G = (V, E)$ với tập hợp đỉnh $V = \{1, 2,..., n\}$, ta cần tìm xem có một lộ trình trong G từ i đến j với tất cả các cặp đỉnh $i, j \in V$ hay không. ***Bao đóng bắc cầu*** của G được định nghĩa dưới dạng đồ thị $G^* = (V, E^*)$, ở đó

$E^* = \{(i, j)$: có một lộ trình từ đỉnh i đến đỉnh j trong $G\}$.

Một cách để tính toán bao đóng bắc cầu của một đồ thị trong $\Theta(n^3)$ thời gian đó là gán một trọng số 1 cho mỗi cạnh của E và chạy thuật toán Floyd-Warshall. Nếu có một lộ trình từ đỉnh i đến đỉnh j, ta có $d_{ij} < n$. Bằng không, ta có $d_{ij} = \infty$.

Có một cách khác tương tự để tính toán bao đóng bắc cầu của G trong $\Theta(n^3)$ thời gian có thể tiết kiệm thời gian và không gian trong thực tế. Phương pháp này bao hàm việc dùng các phép toán lôgic $\vee$ và $\wedge$ thay cho các phép toán số học min và + trong thuật toán Floyd-Warshall. Với $i, j, k = 1, 2,..., n$, ta định nghĩa $t_{ij}^{(k)}$ là 1 nếu ở đó tồn tại một lộ trình trong đồ thị G từ đỉnh i đến đỉnh j với tất cả các đỉnh trung gian trong tập hợp $\{1, 2,..., k\}$, mà bằng không là 0. Ta kiến tạo bao đóng bắc cầu $G^* = (V, E^*)$ bằng cách đặt cạnh (i, j) vào E^* nếu và chỉ nếu $t_{ij}^{(k)} = 1$. Một định nghĩa đệ quy về $t_{ij}^{(k)}$, tương tự như phép truy toán (26.5), là

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{nếu } i \neq j \text{ và } (i, j) \notin E, \\ 1 & \text{nếu } i = j \text{ hoặc } (i, j) \in E, \end{cases}$$

và với $k \geq 1$,

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right). \qquad (26.8)$$

Như trong thuật toán Floyd-Warshall, ta tính toán các ma trận $T^{(k)} = (t_{ij}^{(k)})$ theo thứ tự của k tăng dần.

```
TRANSITIVE-CLOSURE(G)
n ← |V[G]|
for i ← 1 to n
     do for j ← 1 to n
            do if i = j hoặc (i, j) ∈ E[G]
                  then t_ij^(0) ← 1
                  else t_ij^(0) ← 0
for k ← 1 to n
     do for i ← 1 to n
            do for j ← 1 to n
                  do t_ij^(k) ← t_ij^(k-1) ∨ (t_ik^(k-1) ∧ t_kj^(k-1)).
trả về T^(n)
```

Hình 26.5 nêu các ma trận $T^{(k)}$ đã được thủ tục TRANSITIVE-CLOSURE tính toán trên một đồ thị mẫu. Cũng như thuật toán Floyd-Warshall, thời gian thực hiện của thủ tục TRANSITIVE-CLOSURE là $\Theta(n^3)$. Tuy nhiên, trên vài máy tính, các phép toán lôgic trên các giá trị bit đơn thi hành nhanh hơn các phép toán số học trên các từ dữ liệu số nguyên. Hơn nữa, do thuật toán bao đóng bắc cầu trực tiếp chỉ sử dụng các giá trị Bool thay vì các giá trị số nguyên, yêu cầu không gian của nó nhỏ hơn của thuật toán Floyd-Warshall theo một hệ số tương ứng với kích cỡ của một từ của kho lưu trữ máy tính.

Trong Đoạn 26.4, ta sẽ thấy sự tương ứng giữa FLOYD-WARSHALL và TRANSITIVE-CLOSURE tỏ ra trùng khớp hơn. Cả hai thuật toán đều dựa trên một kiểu của cấu trúc đại số có tên "nửa vành đóng."

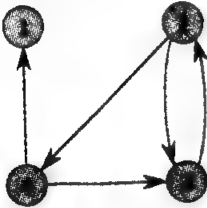
Bài tập

26.2-1

Chạy thuật toán Floyd-Warshall trên đồ thị có hướng gia trọng của Hình 26.2. Nêu ma trận $D^{(k)}$ là kết quả của mỗi lần lặp lại của vòng lặp phía ngoài.

26.2-2

Như đã nêu trên đây, thuật toán Floyd-Warshall yêu cầu $\Theta(n^3)$ không gian, bởi ta tính toán $d_{ij}^{(k)}$ với $i, j, k = 1, 2, ..., n$. Chứng tỏ tính đúng đắn của thủ tục dưới đây, đơn giản thải tất cả các con chữ trên [superscripts], và như vậy chỉ cần $\Theta(n^2)$ không gian.



$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Hình 26.5 Một đồ thị có hướng và các ma trận $T^{(k)}$ được tính toán bởi thuật toán bao đóng bắc cầu.

```
FLOYD-WARSHALL'(W)
n ← rows[W]
D ← W
for k ← 1 to n
     do for i ← 1 to n
          do for j ← 1 to n
               d_ij ← min (d_ij, d_ik + d_kj)
return D
```

26.2-3

Sửa đổi thủ tục FLOYD-WARSHALL để gộp phép tính của các ma trận $\Pi^{(k)}$ theo các phương trình (26.6) và (26.7). Chứng minh chính xác rằng với tất cả $i \in V$, đồ thị con phần tử tiền vị $G_{\pi,i}$ là một cây các lộ trình ngắn nhất có gốc i. (*Mách nước:* Để chứng tỏ $G_{\pi,i}$ là phi chu trình, đầu

tiên ta chứng tỏ $\pi_{ij}^{(k)} = l$ hàm ý $d_{ij}^{(k)} \geq d_{il}^{(k-1)} + w_{lj}$. Sau đó, thích ứng phần chứng minh của Bổ đề 25.8.)

26.2-4

Giả sử ta sửa đổi cách thức điều quản đẳng thức trong phương trình (26.7):

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{nếu } d_{ij}^{(k-1)} < d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{nếu } d_{ij}^{(k-1)} \geq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases}$$

Cách định nghĩa này của ma trận phần tử tiền vị Π có đúng không?

26.2-5

Có thể dùng kết xuất của thuật toán Floyd-Warshall như thế nào để phát hiện sự hiện diện của một chu trình trọng số âm?

26.2-6

Một cách khác để kiến tạo lại các lộ trình ngắn nhất trong thuật toán Floyd-Warshall sử dụng các giá trị $\phi_{ij}^{(k)}$ với $i, j, k = 1, 2, ..., n$, ở đó $\phi_{ij}^{(k)}$ là đỉnh trung gian được đánh số cao nhất của một lộ trình ngắn nhất từ i đến j. Nêu một cách trình bày đệ quy với $\phi_{ij}^{(k)}$, sửa đổi thủ tục FLOYD-WARSHALL để tính toán các giá trị $\phi_{ij}^{(k)}$, và viết lại thủ tục PRINT-ALL-PAIRS-SHORTEST-PATH để sử dụng ma trận $\Phi = (\phi_{ij}^{(n)})$ làm nhập liệu. Ma trận Φ giống bảng s trong bài toán nhân xích ma trận của Đoạn 16.1 như thế nào?

26.2-7

Nêu một thuật toán $O(V, E)$ thời gian để tính toán bao đóng bắc cầu của một đồ thị có hướng $G = (V, E)$.

26.2-8

Giả sử bao đóng bắc cầu của một đồ thị phi chu trình có hướng có thể được tính toán trong $f(V, E)$ thời gian, ở đó $f(V, E) = \Omega(V + E)$ và f tăng đơn điệu. Chứng tỏ thời gian để tính toán bao đóng bắc cầu của một đồ thị chung có hướng là $O(f(V, E))$.

26.3 Thuật toán Johnson cho đồ thị thưa

Thuật toán Johnson tìm các lộ trình ngắn nhất giữa tất cả cặp trong $O(V^2 \lg V + VE)$ thời gian; như vậy đối với đồ thị thưa, nó tốt hơn kiểu bình phương lặp lại của các ma trận hoặc thuật toán Floyd-Warshall theo

tiệm cận. Thuật toán trả về một ma trận các trọng số lộ trình ngắn nhất với tất cả cặp hoặc báo cáo đồ thị nhập liệu chứa một chu trình trọng số âm. Thuật toán Johnson sử dụng cả thuật toán Dijkstra lẫn thuật toán Bellman-Ford làm chương trình con, đã được mô tả trong Chương 25.

Thuật toán Johnson sử dụng kỹ thuật ***tái gia trọng*** [reweighting], làm việc như sau. Nếu tất cả các trọng số cạnh w trong một đồ thị $G = (V, E)$ không âm, ta có thể tìm các lộ trình ngắn nhất giữa tất cả cặp đỉnh bằng cách chạy thuật toán Dijkstra một lần từ mỗi đỉnh; với hàng đợi ưu tiên đống Fibonacci, thời gian thực hiện của thuật toán mọi cặp này là $O(V^2 \lg V + VE)$. Nếu G có các cạnh trọng số âm, ta đơn giản tính toán một tập hợp mới gồm các trọng số cạnh không âm cho phép ta sử dụng cùng phương pháp. Tập hợp mới gồm các trọng số cạnh $\hat{w}$ phải thỏa hai tính chất quan trọng.

1. Với tất cả cặp đỉnh $u, v \in V$, một lộ trình ngắn nhất từ u đến v dùng hàm trọng số w cũng là một lộ trình ngắn nhất từ u đến v dùng hàm trọng số $\hat{w}$.

2. Với tất cả các cạnh (u, v), trọng số mới $\hat{w}(u, v)$ là không âm.

Như sẽ thấy dưới đây, tiến trình tiền xử lý của G để xác định hàm trọng số mới $\hat{w}$ có thể được thực hiện trong $O(VE)$ thời gian.

Bảo toàn các lộ trình ngắn nhất bằng cách tái gia trọng

Như bổ đề dưới đây nêu rõ, ta dễ dàng nảy ra một phép tái gia trọng của các cạnh thỏa tính chất đầu tiên trên đây. Ta dùng δ để thể hiện các trọng số lộ trình ngắn nhất phái sinh từ hàm trọng số w và $\hat{\delta}$ thể hiện các trọng số lộ trình ngắn nhất phái sinh từ hàm trọng số $\hat{w}$.

Bổ đề 26.1 (Tái gia trọng không làm thay đổi các lộ trình ngắn nhất)

Cho một đồ thị có hướng gia trọng $G = (V, E)$ với hàm trọng số $w: E \to \mathbf{R}$, cho $h : V \to \mathbf{R}$ là một hàm bất kỳ ánh xạ các đỉnh theo các số thực. Với mỗi cạnh $(u, v) \in E$, ta định nghĩa

$$\hat{w}(u, v) = w(u, v) + h(u) - h(v) . \qquad (26.9)$$

Cho $p = \langle v_0, v_1, \ldots, v_k \rangle$ là một lộ trình từ đỉnh v_0 đến đỉnh v_k. Sau đó, $w(p) = \delta(v_0, v_k)$ nếu và chỉ nếu $\hat{w}(p) = \hat{\delta}(v_0, v_k)$. Ngoài ra, G có một chu trình trọng số âm dùng hàm trọng số w nếu và chỉ nếu G có một chu trình trọng số âm dùng hàm trọng số $\hat{w}$.

Chứng minh Ta bắt đầu bằng cách chứng tỏ

$$\hat{w}(p) = w(p) + h(v_0) - h(v_k) . \qquad (26.10)$$

Ta có

$$\begin{aligned}\hat{w}(p) &= \sum_{i=1}^{k} \hat{w}(v_{i-1}, v_i) \\ &= \sum_{i=1}^{k} (w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)) \\ &= \sum_{i=1}^{k} w(v_{i-1}, v_i) + h(v_0) - h(v_k) \\ &= w(p) + h(v_0) - h(v_k) .\end{aligned}$$

Đẳng thức thứ ba là do tổng lồng gọn [telescoping sum] trên dòng thứ hai. Giờ đây, bằng sự mâu thuẫn ta chứng tỏ $w(p) = \delta(v_0, v_k)$ hàm ý $\hat{w}(p) = \delta(v_0, v_k)$. Giả sử có một lộ trình p' ngắn hơn từ v_0 đến v_k dùng hàm trọng số $\hat{w}$. Sau đó, $w(p') < w(p)$. Theo phương trình (26.10),

$$\begin{aligned}\hat{w}(p') + h(v_0) - h(v_k) &= \hat{w}(p') \\ &< \hat{w}(p) \\ &= w(p) + h(v_0) - h(v_k) ,\end{aligned}$$

hàm ý $w(p') < w(p)$. Nhưng điều này mâu thuẫn với giả thiết của chúng ta rằng p là một lộ trình ngắn nhất từ u đến v dùng w. Chứng minh của đảo đề là tương tự.

Cuối cùng, ta chứng tỏ G có một chu trình trọng số âm dùng hàm trọng số w nếu và chỉ nếu G có một chu trình trọng số âm dùng hàm trọng số $\hat{w}$. Xét một chu trình $c = \langle v_0, v_1, \ldots, v_k \rangle$, ở đó $v_0 = v_k$. Theo phương trình (26.10),

$$\begin{aligned}\hat{w}(c) &= w(c) + h(v_0) - h(v_k) \\ &= w(c) ,\end{aligned}$$

và như vậy c có trọng số âm dùng w nếu và chỉ nếu nó có trọng số âm dùng $\hat{w}$.

Tạo các trọng số không âm bằng cách tái gia trọng

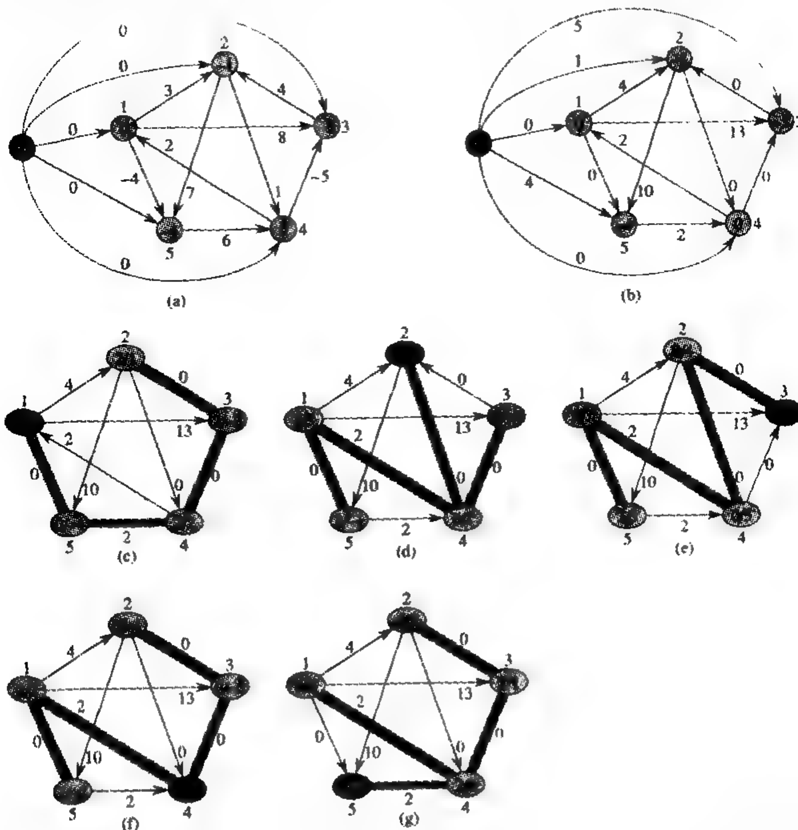
Mục tiêu kế tiếp của chúng ta đó là bảo đảm tính chất thứ hai là đúng: ta muốn $\hat{w}(u, v)$ là không âm với tất cả các cạnh $(u, v) \in E$. Cho một đồ thị có hướng gia trọng $G = (V, E)$ với hàm trọng số $w : E \rightarrow \mathbf{R}$, ta tạo một đồ thị mới $G' = (V', E')$, ở đó $V' = V \cup \{s\}$ với một đỉnh mới $s \notin V$ và $E' = E \cup \{(s, v) : v \in V\}$. Ta mở rộng hàm trọng số w sao cho $w(s, v) = 0$ với tất cả $v \in V$. Lưu ý, bởi s không có các cạnh nhập nó, nên không có lộ trình ngắn nhất trong G', khác với những cạnh có nguồn s, chứa s. Hơn nữa, G' không có chu trình trọng số âm nếu và chỉ nếu G không có các chu trình trọng số âm. Hình 26.6(a) nêu đồ thị G' tương ứng với đồ thị G của Hình 26.1.

Giờ đây giả sử G và G' không có các chu trình trọng số âm. Ta hãy định nghĩa $h(v) = \delta(s, v)$ với tất cả $v \in V'$. Theo Bổ đề 25.3, ta có $h(v) \leq$

$h(u) + w(u, v)$ với tất cả các cạnh $(u, v) \in E'$. Như vậy, nếu ta định nghĩa các trọng số mới w theo phương trình (26.9), ta có $\hat{w}(u,v) = w(u,v) + h(u) - h(v) \geq 0$, và tính chất thứ hai được thỏa. Hình 26.6(b) nêu the đồ thị G' từ Hình 26.6(a) với các cạnh đã tái gia trọng.

Tính toán các lộ trình ngắn nhất mọi cặp

Thuật toán Johnson để tính toán các lộ trình ngắn nhất mọi cặp sử dụng thuật toán Bellman-Ford (Đoạn 25.3) và thuật toán Dijkstra (Đoạn 25.2) làm chương trình con. Nó mặc nhận các cạnh được lưu trữ trong các danh sách kề. Thuật toán trả về ma trận $|V| \times |V|$ thông thường $D = d_{ij}$, ở đó $d_{ij} = \delta(i, j)$, hoặc nó báo cáo đồ thị nhập liệu chứa một chu trình trọng số âm. (Để các chỉ số vào ma trận D có một ý nghĩa nào đó, ta mặc nhận rằng các đỉnh được đánh số từ 1 đến $|V|$.)



Hình 26.6 Thuật toán các lộ trình ngắn nhất mọi cặp của Johnson chạy trên đồ thị của Hình 26.1. **(a)** Đồ thị G' với hàm trọng số ban đầu w. Đỉnh mới s mang màu đen. Trong mỗi đỉnh v là $h(v) = \delta(s, v)$. **(b)** Mỗi cạnh (u, v) được gia trọng lại với hàm trọng số $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$. **(c)-(g)** Kết quả của việc chạy thuật toán Dijkstra trên mỗi đỉnh của G dùng hàm trọng số $\hat{w}$. Trong mỗi phần, đỉnh nguồn u mang màu đen. Trong mỗi đỉnh v là các giá trị $\hat{\delta}(u, v)$ và $\delta(u, v)$, được tách biệt bằng một dấu sổ ngả trước. Giá trị $d_{uv} = \delta(u, v)$ bằng với $\hat{\delta}(u, v) + h \quad - h(u)$.

```
JOHNSON(G)
1   tính toán G', ở đó V[G'] = V[G] ∪ {s} và
        E[G'] = E[G] ∪ {(s,v) : v ∈ V[G]}
2   if BELLMAN-FORD(G', w, s) = FALSE
3      then in "đồ thị nhập liệu chứa một chu trình trọng số âm"
4      else for mỗi đỉnh v ∈ V[G']
5               do ấn định h (v) theo giá trị của δ(s, v)
                        đã được thuật toán Bellman-Ford tính toán
6           for mỗi cạnh (u, v) ∈ E[G']
7               do ŵ (u, v) ← w (u, v) + h(u) - h (v)
8           for mỗi đỉnh u ∈ V[G]
9               do chạy DIJKSTRA(G, ŵ, u) để tính toán
                        δ̂(u,v) với tất cả v ∈ V[G]
10                 for mỗi đỉnh v ∈ V[G]
11                     do d_uv ← δ̂(u, v) + h (v) - h (u)
12  return D
```

Mã này đơn giản thực hiện các hành động mà ta đã định trước đó. Dòng 1 tạo ra G'. Dòng 2 chạy thuật toán Bellman-Ford trên G' với hàm trọng số w. Nếu G', và do đó G, chứa một chu trình trọng số âm, dòng 3 báo cáo bài toán. Các dòng 4-11 mặc nhận rằng G' không chứa các chu trình trọng số âm. Các dòng 4-5 ấn định $h(v)$ theo trọng số lộ trình ngắn nhất $\delta(s, v)$ được thuật toán Bellman-Ford tính toán với tất cả $v \in V'$. Các dòng 6-7 tính toán các trọng số mới $\hat{w}$. Với mỗi cặp đỉnh $u, v \in V$, vòng lặp **for** của các dòng 8-11 tính toán trọng số lộ trình ngắn nhất $\hat{\delta}(u, v)$ bằng cách gọi thuật toán Dijkstra một lần từ mỗi đỉnh trong V. Dòng 11 lưu trữ trong khoản nhập ma trận d_{uv} trọng số lộ trình ngắn nhất đúng đắn $\delta(u, v)$, được phương trình (26.10) tính toán. Cuối cùng, dòng 12 trả về ma trận D hoàn tất. Hình 26.6 nêu tiến trình thi hành thuật toán Johnson.

Ta dễ dàng thấy thời gian thực hiện của thuật toán Johnson là $O(V^2 \lg V + VE)$ nếu hàng đợi ưu tiên trong thuật toán Dijkstra được thực thi bởi một đống Fibonacci. Cách thực thi đống nhị phân đơn giản hơn sẽ mang lại một thời gian thực hiện là $O(V E \lg V)$, mà theo tiệm cận vẫn nhanh hơn thuật toán Floyd-Warshall nếu đồ thị thưa.

Bài tập

26.3-1

Dùng thuật toán Johnson để tìm các lộ trình ngắn nhất giữa tất cả cặp đỉnh trong đồ thị của Hình 26.2. Nêu các giá trị của h và $\hat{w}$ đã tính toán bởi thuật toán.

26.3-2

Nêu mục tiêu của việc bổ sung đỉnh mới s vào V, cho ra V'?

26.3-3

Giả sử $w(u, v) \geq 0$ với tất cả các cạnh $(u, v) \in E$. Đâu là mối quan hệ giữa các hàm trọng số w và $\hat{w}$?

26.4 Một khung sườn chung để giải quyết các bài toán lộ trình trong đồ thị có hướng

Trong đoạn này, ta xét các "nửa vành đóng" [closed semirings], một cấu trúc đại số cho ra a khung sườn chung for giải quyết lộ trình các bài toán trong đồ thị có hướng. Để bắt đầu, ta định nghĩa các nửa vành đóng và mô tả chúng quan hệ như thế nào với một hệ tính các lộ trình có hướng. Sau đó, ta nêu vài ví dụ về các nửa vành đóng và một thuật toán "chung" để tính toán thông tin lộ trình mọi cặp. Cả thuật toán Floyd-Warshall lẫn thuật toán bao đóng bắc cầu trong Đoạn 26.2 đều là những minh dụ hóa về thuật toán chung này.

Định nghĩa về các nửa vành đóng

Một ***nửa vành đóng*** là một hệ thống $(S, \oplus, \odot, \bar{0}, \bar{1})$, ở đó S là một tập hợp các thành phần, $\oplus$(*toán tử tóm tắt*) và $\odot$ (*toán tử mở rộng*) là các phép toán nhị phân trên S, và $\bar{0}$ và $\bar{1}$ là các thành phần của S, thỏa tám tính chất dưới đây:

1. $(S, \oplus, \bar{0})$ là một ***nửa nhóm*** [monoid]:

- S là ***đóng*** dưới $\oplus$: $a \oplus b \in S$ với tất cả $a, b \in S$.
- $\oplus$ là ***kết hợp***: $a \oplus (b \oplus c) = (a \oplus b) \oplus c$ với tất cả $a, b, c \in S$.
- $\bar{0}$ là một ***đồng nhất thức*** với $\oplus$: $a \oplus \bar{0} = \bar{0} \oplus a = a$ với tất cả $a \in S$.

Cũng vậy, $(S, \odot, \bar{1})$ là một nửa nhóm.

2. $\bar{0}$ là một ***"phần tử rỗng"*** [annihilator]: $a \odot \bar{0} = \bar{0} \odot a = \bar{0}$ với tất cả $a \in S$.

3. $\oplus$ là ***giao hoán***: $a \oplus b = b \oplus a$ với tất cả $a, b \in S$.

4. $\oplus$ là ***lũy đẳng*** [idempotent]: $a \oplus a = a$ với tất cả $a \in S$.

5. $\odot$ ***phân phối*** trên $\oplus$: $a \odot (b \oplus c) = (a \odot b) \oplus (a \odot c)$ và $(b \oplus c) \odot a = (b \odot a) \oplus (c \odot a)$ với tất cả $a, b, c \in S$.

6. Nếu $a_1, a_2, a_3, \ldots$ là một dãy các thành phần đếm được của S, thì $a_1 \oplus a_2 \oplus a_3 \oplus \ldots$ được định nghĩa kỹ và nằm trong S.

7. Tính kết hợp, tính giao hoán, và tính lũy đẳng áp dụng cho các tóm tắt vô hạn. (Như vậy, mọi tóm tắt vô hạn có thể được viết lại dưới dạng một tóm tắt vô hạn ở đó mỗi số hạng của tóm tắt được gộp chì một lần và thứ tự đánh giá là tùy ý.)

8. $\odot$ phân phối trên các tóm tắt vô hạn: $a \odot (b_1 \oplus b_2 \oplus b_3 \oplus \ldots) = (a \odot b_1) \oplus (a \odot b_2) \oplus (a \odot b_3) \oplus \ldots$ và $(a_1 \oplus a_2 \oplus a_3 \oplus \ldots) \odot b = (a_1 \odot b) \oplus (a_2 \odot b) \oplus (a_3 \odot b) \oplus \ldots$.

Một hệ tính các lộ trình trong đồ thị có hướng

Mặc dù các tính chất nửa vành đóng có vẻ như trừu tượng, song chúng có thể liên quan đến một hệ tính các lộ trình trong đồ thị có hướng. Giả sử ta có một đồ thị có hướng $G = (V, E)$ và một ***hàm gán nhãn*** $\lambda: V \times V \to S$ ánh xạ tất cả các cặp đỉnh có sắp xếp vào một đồng miền xác định [codomain] S. ***Nhãn cạnh*** [label of edge] $(u, v) \in E$ được ký hiệu là $\lambda(u, v)$. Bởi λ được định nghĩa trên miền xác định $V \times V$, nên nhãn $\lambda(u, v)$ thường xem là $\overline{0}$ nếu (u, v) không phải là một cạnh của G (dưới đây ta sẽ thấy tại sao).

Ta dùng toán tử mở rộng kết hợp $\odot$ để mở rộng khái niệm các nhãn theo các lộ trình. ***Nhãn lộ trình*** $p = <v_1, v_2, \ldots, v_k>$ là

$$\lambda(p) = \lambda(v_1, v_2) \odot \lambda(v_2, v_3) \odot \ldots \odot \lambda(v_{k-1}, v_k).$$

Đồng nhất thức $\overline{1}$ của $\odot$ được dùng làm nhãn lộ trình trống.

Để ví dụ thực hiện của một ứng dụng về các nửa vành đóng, ta dùng các lộ trình ngắn nhất có các trọng số cạnh không âm. Đồng miền xác định S là $\mathbf{R}^{\geq 0} \cup \{\infty\}$, ở đó $\mathbf{R}^{\geq 0}$ là tập hợp các số thực không âm, và $\lambda(i, j) = w_{ij}$ với tất cả $i, j \in V$. Toán tử mở rộng $\odot$ tương ứng với toán tử số học $+$, và do đó nhãn của lộ trình $p = <v_1, v_2, \ldots, v_k>$ là

$$\begin{aligned} \lambda(p) &= \lambda(v_1, v_2) \odot \lambda(v_2, v_3) \odot \ldots \odot \lambda(v_{k-1}, v_k) \\ &= w_{v_1 v_2} + w_{v_2 v_3} + \ldots + w_{v_{k-1} v_k} \\ &= w(p). \end{aligned}$$

Và tất nhiên, vai trò của $\overline{1}$, đồng nhất thức của $\odot$, được chấp nhận bởi 0, đồng nhất thức của $+$. Ta thể hiện lộ trình trống bằng ε, và nhãn của

nó là $\lambda(\varepsilon) = w(\varepsilon) = 0 = 1$.

Do toán tử mở rộng $\odot$ là kết hợp, nên ta có thể định nghĩa nhãn của phép ghép hai lộ trình theo cách tự nhiên. Cho các lộ trình $p_1 = <v_1, v_2, \ldots, v_k>$ và $p_2 = <v_k, v_{k+1}, \ldots, v_l>$, phép ghép của chúng là

$$p_1 \circ p_2 = <v_1, v_2, \ldots, v_k, v_{k+1}, \ldots, v_l>,$$

và nhãn phép ghép của chúng là

$$\begin{aligned}\lambda(p_1 \circ p_2) &= \lambda(v_1, v_2) \odot \lambda(v_2, v_3) \odot \ldots \odot \lambda(v_{k-1}, v_k) \odot \\ &\qquad \lambda(v_k, v_{k+1}) \odot \lambda(v_{k+1}, v_{k+2}) \odot \ldots \odot \lambda(v_{l-1}, v_l) \\ &= (\lambda(v_1, v_2) \odot \lambda(v_2, v_3) \odot \ldots \odot \lambda(v_{k-1}, v_k)) \odot \\ &\qquad (\lambda(v_k, v_{k+1}) \odot \lambda(v_{k+1}, v_{k+2}) \odot \ldots \odot \lambda(v_{l-1}, v_l)) \\ &= \lambda(p_1) \odot \lambda(p_2).\end{aligned}$$

Toán tử tóm tắt $\oplus$, cả giao hoán lẫn kết hợp, được dùng để ***tóm tắt*** các nhãn lộ trình. Nghĩa là, giá trị $\lambda(p_1) \oplus \lambda(p_2)$ cho ra một tóm tắt, ngữ nghĩa học của nó là cụ thể theo ứng dụng, về các nhãn của các lộ trình p_1, và p_2.

Mục tiêu của chúng ta đó là tính toán, với tất cả cặp đỉnh $i, j \in V$, tóm tắt về tất cả các nhãn lộ trình từ i đến j:

$$l_{ij} = \bigoplus_{i \overset{p}{\rightsquigarrow} j} \lambda(p). \qquad (26.11)$$

Ta yêu cầu tính giao hoán và tính kết hợp của $\oplus$ bởi thứ tự ở đó các lộ trình được tóm tắt sẽ không thành vấn đề. Bởi ta dùng phần tử rỗng $\bar{0}$ làm nhãn của một cặp có sắp xếp (u, v) không phải là một cạnh trong đồ thị, nên mọi lộ trình gắng lấy một cạnh vắng mặt sẽ có nhãn $\bar{0}$.

Với các lộ trình ngắn nhất, ta dùng min làm toán tử tóm tắt $\oplus$. Đồng nhất thức của min là ∞, và quả thực ∞ là một phần tử rỗng của +: $a + \infty = \infty + a = \infty$ với tất cả $a \in \mathbf{R}^{\geq 0} \cup \{\infty\}$. Các cạnh vắng mặt có trọng số ∞, và nếu bất kỳ cạnh nào của một lộ trình có trọng số ∞, thì lộ trình cũng vậy.

Ta muốn toán tử tóm tắt $\oplus$ là lũy đẳng, bởi từ phương trình (26.11), ta thấy rằng $\oplus$ sẽ tóm tắt các nhãn của một tập hợp các lộ trình. Nếu p là một lộ trình, thì $\{p\} \cup \{p\} = \{p\}$; nếu ta tóm tắt lộ trình p với chính nó, nhãn kết quả sẽ là nhãn của p: $\lambda(p) \oplus \lambda(p) = \lambda(p)$.

Bởi ta xét các lộ trình có thể không đơn giản, nên có thể có một số lượng vô hạn các lộ trình trong một đồ thị. (Mỗi lộ trình, đơn giản hay không, có một số lượng hữu hạn các cạnh.) Do đó toán tử $\oplus$ có thể áp dụng được cho một số lượng vô hạn các nhãn lộ trình. Nghĩa là, nếu a_1,

a_2, a_3,...là một dãy các thành phần đếm được trong đồng miền xác định S, thì nhãn $a_1 \oplus a_2 \oplus a_3 \oplus ...$ sẽ được định nghĩa kỹ và trong S. Bất kể ta tóm tắt các nhãn lộ trình theo thứ tự nào, và như vậy tính kết hợp và tính giao hoán sẽ áp dụng cho các tóm tắt vô hạn. Vả lại, nếu ta tóm tắt cùng nhãn lộ trình a một số lần vô hạn, ta sẽ có a là kết quả, và như vậy tính lũy đẳng sẽ áp dụng cho các tóm tắt vô hạn.

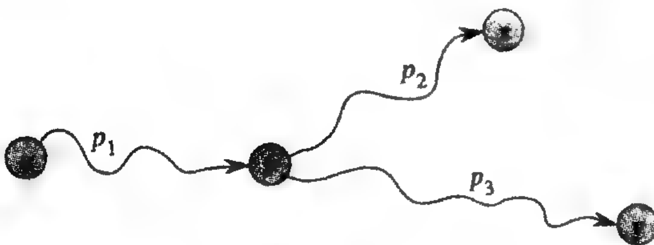
Trở về với ví dụ các lộ trình ngắn nhất, ta hỏi xem liệu min có thể áp dụng được cho một dãy vô hạn các giá trị trong $\mathbf{R}^{\geq 0} \cup \infty$ hay không. Ví dụ, giá trị của $\min_{k=1}^{\infty}\{1/k\}$ có được định nghĩa kỹ hay không? Là có, nếu ta nghĩ toán tử min thực tế trả về cận dưới lớn nhất (cận dưới đúng [infimum]) của các đối số của nó, trong trường hợp đó ta có $\min_{k=1}^{\infty}\{1/k\} = 0$.

Để tính các nhãn phân kỳ các lộ trình, ta cần tính phân phối của toán tử mở rộng $\odot$ trên toán tử tóm tắt $\oplus$. Như đã nêu trong Hình 26.7, giả sử rằng ta có các lộ trình $u \overset{p_1}{\leadsto} v$, $v \overset{p_2}{\leadsto} x$, và $v \overset{p_3}{\leadsto} y$. Với tính phân phối, ta có thể tóm tắt các nhãn của các lộ trình $p_1 \circ p_2$ và $p_1 \circ p_3$ bằng cách tính toán $(\lambda(p_1) \odot \lambda(p_2)) \oplus (\lambda(p_1) \odot \lambda(p_3))$ hoặc $\lambda(p_1) \odot (\lambda(p_2) \oplus \lambda(p_3))$

Bởi có thể có một số lượng vô hạn các lộ trình trong một đồ thị, nên $\odot$ sẽ phân phối trên các tóm tắt vô hạn cũng như hữu hạn. Ví dụ, Hình 26.8 chứa các lộ trình $u \overset{p_1}{\leadsto} v$ và $v \overset{p_2}{\leadsto} x$, cùng với chu trình $v \overset{c}{\leadsto} v$.

Ta phải có khả năng tóm tắt các lộ trình $p_1 \circ p_2$, $p_1 \circ c \circ p_2$, $p_1 \circ c \circ c \circ p_2$,.... Tính phân phối của $\odot$ trên các tóm tắt vô hạn sẽ cho ta

$$\begin{aligned}
&(\lambda(p_1) \odot \lambda(p_2)) \oplus (\lambda(p_1) \odot \lambda(c) \odot \lambda(p_2)) \\
&\quad \oplus (\lambda(p_1) \odot \lambda(c) \odot \lambda(c) \odot \lambda(p_2)) \oplus ... \\
&= \lambda(p_1) \odot (\lambda(p_2) \oplus (\lambda(c) \odot \lambda(p_2)) \oplus (\lambda(c) \odot \lambda(c) \odot \lambda(p_2)) \oplus ...) \\
&= \lambda(p_1) \odot (1 \oplus c \oplus (c \odot c) \oplus (c \odot c \odot c) \oplus ...) \odot \lambda(p_2).
\end{aligned}$$



Hình 26.7 Dùng tính phân phối của $\odot$ trên $\oplus$. Để tóm tắt các nhãn của các lộ trình $p_1 \circ p_2$ và $p_1 \circ p_3$, ta có thể tính toán $(\lambda(p_1) \odot \lambda(p_2)) \oplus (\lambda(p_1) \odot \lambda(p_3))$ hoặc $\lambda(p_1) \odot (\lambda(p_2) \oplus \lambda(p_3))$.



Hình 26.8 Tính phân phối của ⊙ trên các tóm tắt vô hạn của ⊕. Do chu trình c, nên ta có một số lượng vô hạn các lộ trình từ đỉnh v đến đỉnh x. Ta phải có khả năng tóm tắt các lộ trình $p_1 \circ p_2$, $p_1 \circ c \circ p_2$, $p_1 \circ c \circ c \circ p_2$,....

Ta dùng một hệ ký hiệu đặc biệt để thể hiện nhãn của một chu trình có thể được đi ngang qua với một số lần bất kỳ. Giả sử ta có một chu trình c với nhãn $\lambda(c) = a$. Ta có thể băng ngang c zero lần với một nhãn của $\lambda(\varepsilon) = \bar{1}$, một lần với một nhãn của $\lambda(c) = a$, hai lần với một nhãn của $\lambda(c) \odot \lambda(c) = a \odot a$, và vân vân. Nhãn mà ta có bằng cách tóm tắt số lượng vô hạn các lần băng ngang này của chu trình c chính là ***bao đóng*** của a, được định nghĩa bởi

$$a^* = \bar{1} \oplus a \oplus (a \odot a) \oplus (a \odot a \odot a) \oplus (a \odot a \odot a \odot a) \oplus \ldots$$

Như vậy, trong Hình 26.8, ta muốn tính toán $\lambda(p_1) \odot (\lambda(c))^* \odot \lambda(p_2)$.

Với ví dụ các lộ trình ngắn nhất, với một số thực không âm bất kỳ $a \in \mathbf{R}^{\geq 0} \cup \{\infty\}$,

$$\begin{aligned} a^* &= \min_{k=0}^{\infty} \{ka\} \\ &= 0. \end{aligned}$$

Sự diễn dịch của tính chất này đó là bởi vì tất cả các chu trình đều có trọng số không âm, nên không bao giờ cần lộ trình ngắn nhất để băng ngang nguyên cả một chu trình.

Các ví dụ về các nửa vành đóng

Ta đã thấy một ví dụ về một nửa vành đóng, tức là $S_1 = (\mathbf{R}^{\geq 0} \cup \{\infty\}, \min, +, \infty, 0)$, mà ta đã dùng cho các lộ trình ngắn nhất với các trọng số cạnh không âm. (Như đã nêu trên đây, toán tử min thực tế trả về cận dưới lớn nhất của các đối số của nó.) Ta cũng đã chứng tỏ $a^* = 0$ với tất cả $a \in \mathbf{R}^{\geq 0} \cup \{\infty\}$.

Tuy nhiên, ta đã đoan chắc rằng cho dù có các trọng số cạnh âm, thuật toán Floyd-Warshall vẫn tính toán các trọng số lộ trình ngắn nhất miễn là không có các chu trình trọng số âm nào hiện diện. Nhờ cộng toán tử bao đóng thích hợp và mở rộng đồng miền xác định của các nhãn theo $\mathbf{R} \cup \{-\infty, +\infty\}$, nên ta có thể tìm một nửa vành đóng để điều quản các chu trình trọng số âm. Dùng min với ⊕ và + với ⊙, người đọc có thể xác minh rằng bao đóng của một $a \in \mathbf{R} \cup \{-\infty, +\infty\}$ là

$$a^* = \begin{cases} 0 & \text{nếu } a \geq 0, \\ -\infty & \text{nếu } a < 0. \end{cases}$$

Trường hợp thứ hai ($a < 0$) mô hình hóa tình huống ở đó ta có thể băng ngang một chu trình trọng số âm với số lần vô hạn để được một trọng số của $-\infty$ trên bất kỳ lộ trình nào chứa chu trình. Như vậy, nửa vành đóng để sử dụng cho thuật toán Floyd-Warshall với các trọng số cạnh âm là $S_2 = (\mathbf{R} \cup \{-\infty, +\infty\}, \min, +, +\infty, 0)$. (Xem Bài tập 26.4-3.)

Với bao đóng bắc cầu, ta dùng nửa vành đóng $S_3 = (\{0, 1\}, \vee, \wedge, 0, 1)$, ở đó $\lambda(i, j) = 1$ nếu $(i, j) \in E$, và bằng không $\lambda(i, j) = 0$. Vậy ta có $0^* = 1^* = 1$.

Một thuật toán lập trình động cho các nhãn lộ trình có hướng

Giả sử ta có một đồ thị có hướng $G = (V, E)$ với hàm gán nhãn $\lambda: V \times V \rightarrow S$. Các đỉnh được đánh số từ 1 qua n. Với mỗi cặp đỉnh $i, j \in V$, ta muốn tính toán phương trình (26.11):

$$l_{ij} = \bigoplus_{i \overset{p}{\leadsto} j} \lambda(p),$$

là kết quả của việc tóm tắt tất cả các lộ trình từ i đến j dùng toán tử tóm tắt $\oplus$. Ví dụ, với các lộ trình ngắn nhất, ta muốn tính toán

$$l_{ij} = \delta(i,j) = \min_{i \overset{p}{\leadsto} j} \{w(p)\}.$$

Có một thuật toán lập trình động để giải quyết bài toán này, và dạng của nó cũng tương tự như thuật toán Floyd-Warshall và thuật toán bao đóng bắc cầu. Cho $Q_{ij}^{(k)}$ là tập hợp các lộ trình từ đỉnh i đến đỉnh j với tất cả các đỉnh trung gian trong tập hợp $\{1, 2, ..., k\}$. Ta định nghĩa

$$l_{ij}^{(k)} = \bigoplus_{p \in Q_{ij}^{(k)}} \lambda(p).$$

Lưu ý tính tương tự đối với các phần định nghĩa của $d_{ij}^{(k)}$ trong thuật toán Floyd-Warshall và $t_{ij}^{(k)}$ trong thuật toán bao đóng bắc cầu. Ta có thể định nghĩa l_{ij}^{k} một cách đệ quy bằng

$$l_{ij}^{(k)} = l_{ij}^{(k-1)} \oplus \left(l_{ik}^{(k-1)} \odot (l_{kk}^{(k-1)})^* \odot l_{kj}^{(k-1)} \right). \quad (26.12)$$

Phép truy toán (26.12) gợi lại các phép truy toán (26.5) và (26.8), nhưng có gộp thêm một thừa số của $(l_{kk}^{(k-1)})^*$. Thừa số này biểu diễn phép tóm tắt của tất cả các chu trình đi qua đỉnh k và có tất cả các đỉnh khác trong tập hợp $\{1, 2, ..., k-1\}$. (Khi ta mặc nhận không có các chu trình trọng số âm trong thuật toán Floyd-Warshall, $(c_{kk}^{(k-1)})^*$ là 0, tương

ứng với $\bar{1}$, trọng số của một chu trình trống. Trong thuật toán bao đóng bắc cầu, lộ trình trống từ k đến k cho ta $(l_{kk}^{(k-1)})^* = 1 = \bar{1}$. . Như vậy, với cả hai thuật toán này, ta có thể bỏ qua thừa số của $(l_{kk}^{(k-1)})^*$, bởi nó chính là đồng nhất thức của ⊙) Cơ sở của phần định nghĩa đệ quy là

$$l_{ij}^{(0)} = \begin{cases} \lambda(i,j) & \text{nếu } i \neq j \\ \bar{1} \oplus \lambda(i,j) & \text{nếu } i = j, \end{cases}$$

mà ta có thể thấy như sau. Nhãn của lộ trình một cạnh $\langle i, j \rangle$ đơn giản là $\lambda(i,j)$ (bằng với $\bar{0}$ nếu (i,j) không phải lá một cạnh trong E). Ngoài ra, nếu $i = j$, thì $\bar{1}$ là nhãn của lộ trình trống từ i đến i.

Thuật toán lập trình động tính toán các giá trị $l_{ij}^{(k)}$ theo thứ tự của k tăng dần. Nó trả về ma trận $L^{(n)} = (l_{ij}^{(n)})$.

```
COMPUTE-SUMMARIES(λ, V)
n ← |V|
for i ← 1 to n
    do for j ← 1 to n
        do if i = j
            then l_ij^(0) ← 1̄ ⊕ λ(i, j)
            else l_ij^(0) ← λ(i, j)
for k ← 1 to n
    do for i ← 1 to n
        do for j ← 1 to n
            do l_ij^(k) ← l_ij^(k-1) ⊕ (l_ik^(k-1) ⊙ (l_kk^(k-1))* ⊙ l_kj^(k-1)). ⊙
return L^(n)
```

Thời gian thực hiện của thuật toán này tùy thuộc vào thời gian tính toán ⊙, ⊕, và *. Nếu ta cho $T_{\odot}$, $T_{\oplus}$, và T_* biểu diễn các thời gian này, thì thời gian thực hiện của COMPUTE-SUMMARIES là $\Theta(n^3(T_{\odot} + T_{\oplus} + T_*))$, là $\Theta(n^3)$ nếu mỗi trong số ba phép toán mất $O(1)$ thời gian.

Bài tập

26.4-1

Xác minh $S_1 = (\mathbf{R}^{\geq 0} \cup \{\infty\}, \min, +, \infty, 0)$ và $S_3 = (\{0, 1\}, \vee, \wedge, 0, 1)$ là các nửa vành đóng.

26.4-2

Xác minh $S_2 = (\mathbf{R} \cup \{-\infty, +\infty\}, \min, +, +\infty, 0)$ là một nửa vành đóng.

Nêu giá trị của $a + (-\infty)$ với $a \in \mathbf{R}$? Còn $(-\infty) + (+\infty)$ thì sao?

26.4-3

Viết lại thủ tục COMPUTE-SUMMARIES để sử dụng nửa vành đóng S_2, sao cho nó thực thi thuật toán Floyd-Warshall. Giá trị của $-\infty + \infty$ phải là gì?

26.4-4

Hệ thống $S_4 = (\mathbf{R}, +, \cdot, 0, 1)$ có phải là một nửa vành đóng không?

26.4-5

Ta có thể dùng một nửa vành đóng tùy ý cho thuật toán Dijkstra không? Với thuật toán Bellman-Ford thì sao? Với thủ tục FASTER-ALL-PAIRS-SHORTEST-PATHS thì sao?

26.4-6

Một hãng vận chuyển muốn gửi một xe tải từ Castroville đến Boston chở đầy cây atisô càng nặng càng tốt, nhưng mỗi tuyến đường ở nước Mỹ có một giới hạn trọng số cực đại đối với các xe tải sử dụng đường. Lập mô hình bài toán này với một đồ thị có hướng $G = (V, E)$ và một nửa vành đóng thích hợp, đồng thời đưa ra một thuật toán hiệu quả để giải quyết nó.

Các Bài Toán

26-1 Bắc cầu bao đóng của một đồ thị động

Giả sử rằng ta muốn duy trì bao đóng bắc cầu của một đồ thị có hướng $G = (V, E)$ khi ta chèn các cạnh vào E. Nghĩa là, sau khi mỗi cạnh đã được chèn, ta muốn cập nhật bao đóng bắc cầu của các cạnh được chèn cho đến lúc đó. Giả sử thoạt đầu đồ thị G không có các cạnh và bao đóng bắc cầu phải được biểu diễn dưới dạng một ma trận bool.

a. Nêu cách cập nhật bao đóng bắc cầu $G^* = (V, E^*)$ của một đồ thị $G = (V, E)$ trong $O(V^2)$ thời gian khi một cạnh mới được bổ sung vào G.

b. Nêu một ví dụ về một đồ thị G và một cạnh e sao cho chỉ cần $\Omega(V^2)$ thời gian để cập nhật bao đóng bắc cầu sau phép chèn của e vào G.

c. Mô tả một thuật toán hiệu quả để cập nhật bao đóng bắc cầu khi các cạnh được chèn vào đồ thị. Với một dãy n phép chèn, thuật toán của bạn sẽ chạy trong tổng thời gian $\sum_{i=1}^{n} t_i = O(V^3)$, ở đó t_i là thời gian để cập nhật bao đóng bắc cầu khi cạnh thứ i được chèn. Chứng minh

thuật toán của bạn đạt cận thời gian này.

26-2 Các lộ trình ngắn nhất trong đồ thị trù mật ϵ

Một đồ thị $G = (V, E)$ là ***trù mật*** ε nếu $|E| = \Theta(V^{1+\varepsilon})$ với một hằng ε trong miền giá trị $0 < \varepsilon \leq 1$. Nhờ dùng các đống thuộc d (xem Bài toán 7-2) trong các thuật toán các lộ trình ngắn nhất trên các đồ thị trù mật ε, ta có thể so khớp ε - thời gian thực hiện của các thuật toán gốc đống Fibonacci mà không dùng một cấu trúc dữ liệu phức tạp như vậy.

a. Nêu các thời gian thực hiện tiệm cận của INSERT, EXTRACT-MIN, và DECREASE-KEY, dưới dạng một hàm của d và số n thành phần trong một đống thuộc d? Nêu các thời gian thực hiện này nếu ta chọn $d = \Theta(n^{\alpha})$ với một hằng $0 < \alpha \leq 1$? So sánh các thời gian thực hiện này với các mức hao phí khấu trừ của các phép toán này đối với một đống Fibonacci.

b. Nêu cách tính các lộ trình ngắn nhất từ một nguồn đơn trên một đồ thị trù mật ε có hướng $G = (V, E)$ không có các trọng số cạnh âm trong $O(E)$ thời gian. (*Mách nước:* Chọn d làm một hàm của ε.)

c. Nêu cách giải quyết bài toán các lộ trình ngắn nhất mọi cặp trên một đồ thị trù mật ε có hướng $G = (V, E)$ không có các trọng số cạnh âm trong $O(V, E)$ thời gian.

d. Nêu cách giải quyết bài toán các lộ trình ngắn nhất mọi cặp trong $O(VE)$ thời gian trên một đồ thị trù mật ε có hướng $G = (V, E)$ có thể có các trọng số cạnh âm nhưng không có các chu trình trọng số âm.

26-3 Cây tỏa nhánh cực tiểu dưới dạng một nửa vành đóng

Cho $G = (V, E)$ là một đồ thị không hướng, liên thông với hàm trọng số $w : E \rightarrow \mathbf{R}$. Cho tập hợp đỉnh là $V = \{1, 2, \ldots, n\}$, ở đó $n = |V|$, và mặc nhận rằng tất cả các trọng số cạnh $w(i, j)$ là duy nhất. Cho T là cây tỏa nhánh cực tiểu duy nhất của G (xem Bài tập 24.1-6). Trong bài toán này, ta sẽ xác định T bằng cách dùng một nửa vành đóng, như B. M. Maggs và S. A. Plotkin đã đề xuất. Trước tiên, ta xác định, với mỗi cặp đỉnh $i, j \in V$, trọng số ***minimax***

$$m_{ij} = \min_{i \overset{p}{\leadsto} j} \ \max_{\text{các cạnh } e \text{ trên } p} w(e).$$

a. Xác minh ngắn gọn sự quả quyết rằng $S = (\mathbf{R} \cup \{-\infty, \infty\}, \min, \max, \infty, -\infty)$ là một nửa vành đóng.

Bởi S là một nửa vành đóng, ta có thể dùng thủ tục COMPUTE-SUMMARIES để xác định các trọng số minimax m_{ij} trong đồ thị G. Cho m_{ij}^{k} là trọng số minimax trên tất cả các lộ trình từ đỉnh i đến đỉnh j với tất cả các đỉnh trung gian trong tập hợp $\{1, 2, \ldots, k\}$.

b. Nêu một phép truy toán với $m_{ij}^{(k)}$, ở đó $k \geq 0$.

c. Cho $T_m = \{(i, j) \in E : w(i, j) = m_{ij}\}$. Chứng minh các cạnh trong T_m hình thành một cây tỏa nhánh của G.

d. Chứng tỏ $T_m = T$. (*Mách nước:* Xét hiệu ứng của việc bổ sung cạnh (i, j) vào T và gỡ bỏ một cạnh trên một lộ trình khác từ i đến j. Cũng xét hiệu ứng của việc gỡ bỏ cạnh (i, j) ra khỏi T và thay nó bằng một cạnh khác.)

Ghi chú Chương

Lawler [132] có đề cập chi tiết về bài toán các lộ trình ngắn nhất mọi cặp, mặc dù ông không phân tích các giải pháp cho đồ thị thưa. Ông quy thuật toán phép nhân ma trận cho văn hóa dân gian. Thuật toán Floyd-Warshall là của Floyd [68], ông đặt nó trên cơ sở một định lý của Warshall [198] mô tả cách tính toán bao đóng bắc cầu của các ma trận bool. Cấu trúc đại số nửa vành đóng xuất hiện trong Aho, Hopcroft, và Unman [4]. Thuật toán Johnson xuất xứ từ [114].

27 Luồng Cực Đại

Giống như trường hợp lập mô hình một bản đồ đường xá dưới dạng một đồ thị có hướng để tìm ra lộ trình ngắn nhất từ điểm này đến điểm kia, ta cũng có thể diễn dịch một đồ thị có hướng dưới dạng một "mạng luồng" và dùng nó để giải đáp các câu hỏi về các luồng vật chất [material flows]. Hãy tưởng tượng một luồng di chuyển vật chất qua một hệ thống từ một nguồn, ở đó vật chất được tạo, đến một bồn [sink], ở đó nó được tiêu thụ. Nguồn tạo ra vật chất theo một tốc độ đều đặn, và bồn tiêu thụ vật chất theo cùng tốc độ. "Luồng lưu chuyển" [flow] của vật chất tại một điểm bất kỳ trong hệ thống theo trực giác là tốc độ mà vật chất di chuyển. Có thể dùng các mạng luồng để lập mô hình các chất lỏng chảy qua các ống dẫn, các linh kiện qua các dây chuyền lắp ráp, dòng điện qua các mạng lưới điện, thông tin qua các mạng truyền thông, và vân vân.

Mỗi cạnh có hướng trong một mạng luồng có thể được xem như một ống dẫn cho vật chất. Mỗi ống dẫn có một dung lượng đã định, được nêu dưới dạng tốc độ cực đại qua đó vật chất có thể lưu chuyển qua ống dẫn, như 200 gallon chất lỏng mỗi giờ qua một ống dẫn hoặc 20 ampere dòng điện qua một dây dẫn. Các đỉnh là các khớp nối của ống dẫn, và khác với nguồn và bồn, vật chất chảy qua các đỉnh mà không ứ đọng lại trong chúng. Nói cách khác, tốc độ mà vật chất nhập một đỉnh phải bằng tốc độ mà nó rời đỉnh. Ta gọi tính chất này là "sự bảo toàn luồng lưu," và nó giống như Luật Dòng Điện [Current Law] của Kirchhoff khi vật chất là dòng điện.

Bài toán luồng cực đại là bài toán đơn giản nhất liên quan đến các mạng luồng. Nó đặt vấn đề, Đâu là tốc độ lớn nhất mà vật chất có thể chuyển đi từ nguồn đến bồn mà không vi phạm bất kỳ hạn chế dung lượng nào? Như sẽ thấy trong chương này, bài toán này có thể được giải quyết bằng các thuật toán hiệu quả. Hơn nữa, ta có thể thích ứng các kỹ thuật căn bản mà các thuật toán này sử dụng để giải quyết các bài toán luồng mạng khác.

Chương này trình bày hai phương pháp chung để giải quyết bài toán luồng cực đại. Đoạn 27.1 trình bày các khái niệm về các mạng luồng và

các luồng lưu chuyển, chính thức định nghĩa bài toán luồng cực đại. Đoạn 27.2 mô tả phương pháp cổ điển của Ford và Fulkerson để tìm các luồng cực đại. Đoạn 27.3 mô tả một ứng dụng của phương pháp này: tìm một so khớp cực đại trong một đồ thị hai nhánh không hướng. Đoạn 27.4 trình bày phương pháp đẩy luồng trước [preflow-push], làm cơ sở cho nhiều thuật toán nhanh nhất để giải quyết các bài toán luồng mạng. Đoạn 27.5 đề cập thuật toán "nâng tới trước" [lift-to-front], một thực thi cụ thể của phương pháp đẩy luồng trước chạy trong thời gian $O(V^3)$. Tuy thuật toán này không phải là thuật toán nhanh nhất được biết, song nó minh họa vài kỹ thuật được dùng trong các thuật toán nhanh nhất theo tiệm cận, và nó tương đối hiệu quả trong thực tế.

27.1 Các mạng luồng

Trong đoạn này, ta nêu một định nghĩa về các mạng luồng theo lý thuyết đồ thị, đề cập các tính chất của chúng, và định nghĩa bài toán luồng cực đại một cách chính xác. Ta cũng giới thiệu một hệ ký hiệu hữu ích.

Các mạng luồng và các luồng

Một ***mạng luồng*** [flow network] $G = (V, E)$ là một đồ thị có hướng ở đó mỗi cạnh $(u, v) \in E$ có một dung lượng không âm $c(u, v) \geq 0$. Nếu $(u, v) \notin E$, ta mặc nhận $c(u, v) = 0$. Ta phân biệt hai đỉnh trong một mạng luồng: một ***nguồn*** [source] s và một ***bồn*** [sink] t. Để tiện dụng, ta mặc nhận rằng mọi đỉnh nằm trên một lộ trình nào đó từ nguồn đến bồn. Nghĩa là, với mọi đỉnh $v \in V$, ta có một lộ trình $s \leadsto v \leadsto t$. Do đó, đồ thị được liên thông, và $|E| \geq |V| - 1$. Hình 27.1 có nêu một ví dụ của một mạng luồng.

Giờ đây, ta đã sẵn sàng để định nghĩa các luồng chính thức hơn. Cho $G = (V, E)$ là một mạng luồng (với một hàm dung lượng mặc định c). Cho s là nguồn của mạng, và cho t là bồn. Một luồng trong G là một hàm có giá trị thực $f : V \times V \to \mathbf{R}$ thỏa ba tính chất dưới đây:

Ràng buộc dung lượng: Với tất cả $u, v \in V$, ta yêu cầu $f(u, v) \leq c(u, v)$.

Tính đối xứng ghềnh: Với tất cả $u, v \in V$, ta yêu cầu $f(u, v) = -f(v, u)$.

Bảo toàn luồng lưu: Với tất cả $u \in V - \{s, t\}$, ta yêu cầu

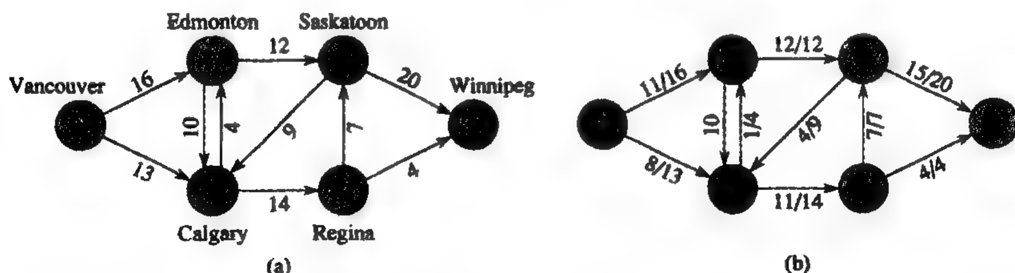
$$\sum_{v \in V} f(u, v) = 0 .$$

Khối lượng $f(u, v)$, có thể là dương hoặc âm, được gọi là ***luồng mạng***

[net flow] từ đỉnh u đến đỉnh v. ***Giá trị*** của một luồng f được định nghĩa là

$$|f| = \sum_{v \in V} f(s, v) , \qquad (27.1)$$

nghĩa là, tổng luồng mạng rời nguồn. (Ở đây, hệ ký hiệu $|\cdot|$ thể hiện giá trị luồng, chứ không phải là giá trị tuyệt đối hoặc bản số.) Trong ***bài toán luồng cực đại***, ta có một mạng luồng G với nguồn s và bồn t, và ta muốn tìm một luồng của giá trị cực đại từ s đến t.



Hình 27.1 (a) Một mạng luồng $G = (V, E)$ cho bài toán vận chuyển của công ty Lucky Puck. Nhà máy Vancouver là nguồn s, và nhà kho Winnipeg là bồn t. Các quả bóng khúc côn cầu được chuyển đi qua các thành phố trung gian, nhưng chỉ $c(u, v)$ thùng hàng mỗi ngày có thể đi từ thành phố u đến thành phố v. Mỗi cạnh được gán nhãn with của nó dung lượng. **(b)** *A flow f* trong G with giá trị *Nếu* $|f| = 19$. Chỉ các mạng luồng được nêu. *Nếu* $f(u, v) > 0$, cạnh (u, v) được gán nhãn bằng $f(u, v)/c(u, v)$. (Hệ ký hiệu dấu sổ (/) được dùng để đơn thuần tách biệt luồng với dung lượng; nó không biểu hiện phép chia.) *Nếu* $f(u,v) \leq 0$, cạnh (u, v) chỉ được gán nhãn bằng dung lượng của nó.

Trước khi xem một ví dụ về bài toán luồng mạng, ta hãy khảo sát ngắn gọn ba tính chất luồng. Sự ràng buộc dung lượng đơn giản muốn nói luồng mạng từ đỉnh này đến đỉnh khác không được vượt quá dung lượng đã cho. Tính đối xứng ghềnh nói rằng luồng mạng từ một đỉnh u đến một đỉnh v là âm của luồng mạng theo hướng nghịch đảo. Như vậy, luồng mạng từ một đỉnh đến chính nó sẽ là 0, bởi với tất cả $u \in V$, ta có $f(u, u) = -f(u, u)$, hàm ý rằng $f(u, u) = 0$. Tính chất bảo toàn luồng lưu nói rằng tổng luồng mạng rời một đỉnh khác với nguồn hoặc bồn sẽ là 0. Theo tính đối xứng ghềnh, ta có thể viết lại tính chất bảo toàn luồng lưu là

$$\sum_{u \in V} f(u, v) = 0 .$$

với tất cả $v \in V - \{s, t\}$. Nghĩa là, tổng luồng mạng vào một đỉnh là 0.

Cũng quan sát thấy có thể không có luồng mạng nào giữa u và v nếu không có cạnh nào giữa chúng. Nếu không có $(u, v) \in E$ cũng không có

$(v, u) \in E$, thì $c(u, v) = c(v, u) = 0$. Do đó, theo sự ràng buộc dung lượng, $f(u, v) \leq 0$ và $f(v, u) \leq 0$. Nhưng bởi $f(u, v) = -f(v, u)$, nên theo tính đối xứng ghềnh, ta có $f(u,v) = f(v,u) = 0$. Như vậy, luồng mạng phi zero từ đỉnh u đến đỉnh v hàm ý rằng $(u, v) \in E$ hoặc $(v, u) \in E$ (hoặc cả hai).

Nhận xét cuối cùng của chúng ta về các tính chất luồng có liên quan đến luồng mạng dương. ***Luồng mạng dương*** nhập một đỉnh v được định nghĩa bởi

$$\sum_{\substack{u \in V \\ f(u,v) > 0}} f(u, v) = 0 . \qquad (27.2)$$

Luồng mạng dương rời một đỉnh được định nghĩa một cách đối xứng. Một diễn dịch về tính chất bảo toàn luồng lưu đó là luồng mạng dương nhập một đỉnh khác với nguồn hoặc bồn phải bằng với luồng mạng dương rời đỉnh.

Một ví dụ về luồng mạng

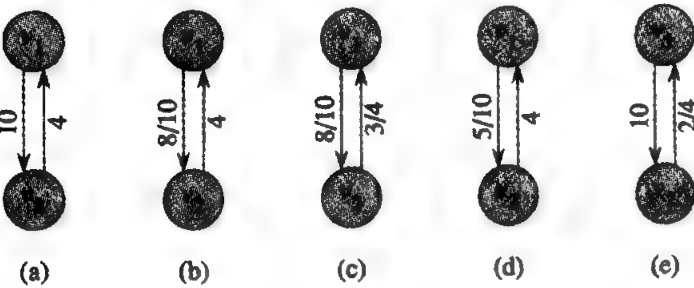
Một mạng luồng [flow network] có thể lập mô hình bài toán vận chuyển nêu trong Hình 27.1. Công ty Lucky Puck có một nhà máy (nguồn s) ở Vancouver chế tạo các quả bóng khúc côn cầu, và có một nhà kho (bồn t) ở Winnipeg lưu trữ chúng. Lucky Puck thuê không gian trên các xe tải từ một hãng khác để vận chuyển các quả bóng khúc côn cầu từ nhà máy đến nhà kho. Do các xe tải di chuyển trên các tuyến đường đã định giữa các thành phố và có một dung lượng hạn chế, Lucky Puck có thể vận chuyển tối đa $c(u, v)$ thùng hàng mỗi ngày giữa mỗi cặp thành phố u và v trong Hình 27.1(a). Lucky Puck có quyền điều khiển đối với các tuyến đường và các dung lượng này và do đó không thể thay đổi mạng luồng nêu trong Hình 27.1(a). Mục tiêu của họ đó là xác định khối lượng lớn nhất p thùng hàng mỗi ngày có thể chuyển đi để rồi tạo ra khối lượng này, bởi không cần gì phải chế tạo số lượng quả bóng khúc côn cầu nhiều hơn mức có thể chuyển chúng đến nhà kho.

Tốc độ mà các quả bóng khúc côn cầu được chuyển đi dọc theo một tuyến đường vận chuyển bất kỳ được xem là một luồng. Các quả bóng khúc côn cầu xuất phát từ nhà máy với tốc độ của p thùng hàng mỗi ngày, và p thùng hàng phải đến nhà kho mỗi ngày. Lucky Puck không quan tâm đến thời gian mà một quả bóng đã cho di chuyển từ nhà máy đến nhà kho; họ chỉ quan tâm rằng p thùng hàng mỗi ngày rời nhà máy và p thùng hàng mỗi ngày đến nhà kho. Các hạn chế dung lượng bị khống chế bởi mức hạn chế mà luồng $f(u, v)$ từ thành phố u đến thành phố v tối đa là $c(u, v)$ thùng hàng mỗi ngày. Trong một trạng thái ổn định, tốc độ mà các quả bóng khúc côn cầu nhập một thành phố trung gian trong mạng vận chuyển phải bằng tốc độ mà chúng rời đi; bằng

không, chúng sẽ ùn đống. Do đó, sự bảo toàn luồng được tuân thủ. Như vậy, một luồng cực đại trong mạng xác định số lượng cực đại p thùng hàng mỗi ngày có thể được chuyển đi.

Hình 27.1 (b) nêu một luồng khả dĩ trong mạng được biểu thị theo một cách tương ứng tự nhiên với các chuyến hàng. Với bất kỳ hai đỉnh u và v trong mạng, luồng mạng $f(u, v)$ tương ứng với một chuyến hàng gồm $f(u, v)$ thùng hàng mỗi ngày từ u đến v. Nếu $f(u, v)$ là 0 hoặc âm, thì không có chuyến hàng nào từ u đến v. Như vậy, trong Hình 27.1(b), chỉ các cạnh có luồng mạng dương mới được nêu, theo sau là một dấu sổ ngả trước và dung lượng của cạnh.

Để dễ hiểu rõ mối quan hệ giữa các luồng mạng và các chuyến hàng, ta tập trung vào các chuyến hàng giữa hai đỉnh. Hình 27.2(a) nêu đồ thị con được cảm sinh bởi các đỉnh v_1 và v_2 trong mạng luồng của Hình 27.1. Nếu Lucky Puck chuyển 8 thùng hàng mỗi ngày từ v_1 đến v_2, kết quả được nêu trong Hình 27.2(b): luồng mạng từ v_1 đến v_2 là 8 thùng hàng mỗi ngày. Theo tính đối xứng ghềnh, ta cũng nói rằng luồng mạng theo hướng ngược lại, từ v_2 đến v_1 là –8 thùng hàng mỗi ngày, cho dù ta không chuyển quả bóng khúc côn cầu nào từ v_2 đến v_1. Nói chung, luồng mạng từ v_1 đến v_2 là số lượng thùng hàng mỗi ngày được chuyển đi từ v_1 đến v_2 trừ số lượng mỗi ngày được chuyển đi từ v_2 đến v_1. Quy ước của chúng ta để biểu thị các luồng mạng đó là chỉ nêu các luồng mạng dương, bởi chúng nêu rõ các chuyến hàng thực tế; như vậy, chỉ một 8 xuất hiện trong hình, mà không có –8 tương ứng.



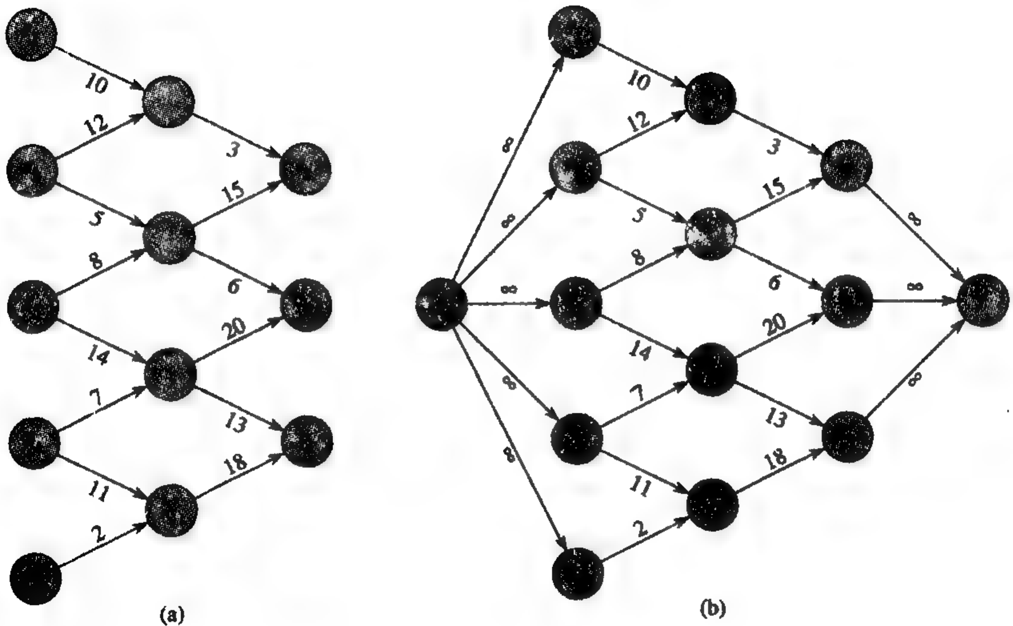
Hình 27.2 Sự khử. **(a)** Các đỉnh v_1 và v_2, với $c(v_1, v_2) = 10$ và $c(v_2, v_1) = 4$. **(b)** Cách nêu rõ luồng mạng khi 8 thùng hàng mỗi ngày được chuyển đi từ v_1 đến v_2. **(c)** Một chuyến hàng bổ sung 3 thùng hàng mỗi ngày được thực hiện từ v_2 đến v_1. **(d)** Bằng cách khử luồng đi theo các hướng ngược lại, ta có thể biểu diễn tình huống trong (c) bằng luồng mạng dương theo chỉ một hướng. **(e)** Một chuyến 7 thùng hàng mỗi ngày khác được chuyển đi từ v_2 đến v_1 .

Giờ đây ta bổ sung một chuyến hàng khác, lần này là 3 thùng hàng mỗi ngày từ v_2 đến v_1. Hình 27.2(c) có nêu phần biểu diễn tự nhiên của kết quả. Giờ đây ta có một tình huống ở đó có các chuyến hàng theo cả hai hướng giữa v_1 và v_2. Ta chuyển 8 thùng hàng mỗi ngày từ v_1 đến v_2 và 3 thùng hàng mỗi ngày từ v_2 đến v_1. Đâu là các luồng mạng giữa hai đỉnh? Luồng mạng từ v_1 đến v_2 là 8 - 3 = 5 thùng hàng mỗi ngày, và luồng mạng từ v_2 các v_1 là 3 – 8 = –5 các thùng hàng mỗi ngày.

Tình huống tương đương trong kết quả của nó với tình huống nêu trong Hình 27.2(d), ở đó 5 thùng hàng mỗi ngày được chuyển đi từ v_1 đến v_2 và không có chuyến hàng nào được thực hiện từ v_2 đến v_1. Trên thực tế, 3 thùng hàng mỗi ngày từ v_2 đến v_1 được ***khử*** bởi 3 trong số 8 thùng hàng mỗi ngày từ v_1 đến v_2. Trong cả hai tình huống, luồng mạng từ v_1 đến v_2 là 5 thùng hàng mỗi ngày, nhưng trong (d), các chuyến hàng thực tế thực hiện theo chỉ một hướng.

Nói chung, phép cho phép ta biểu diễn các chuyến hàng giữa hai thành phố bằng một luồng mạng dương dọc theo tối đa một trong hai cạnh giữa các đỉnh tương ứng. Nếu có luồng mạng zero hoặc âm từ đỉnh này đến đỉnh khác, ta không cần thực hiện chuyến hàng nào theo hướng đó. Nghĩa là, bất kỳ tình huống nào ở đó các quả bóng khúc côn cầu được chuyển đi theo cả hai hướng giữa hai thành phố đều có thể dùng phép hủy để biến đổi thành một tình huống tương đương ở đó các quả bóng khúc côn cầu được chuyển đi theo chỉ một hướng: hướng của luồng mạng dương. Các hạn chế dung lượng không bị vi phạm bởi phép biến đổi này, bởi ta rút gọn các chuyến hàng trong cả hai hướng, và các hạn chế bảo toàn không bị vi phạm, bởi luồng mạng giữa hai đỉnh là giống nhau.

Tiếp tục với ví dụ của chúng ta, hãy xác định hiệu ứng của việc chuyển một chuyến 7 thùng hàng mỗi ngày khác từ v_2 đến v_1. Hình 27.2(e) nêu kết quả dùng quy ước chỉ biểu thị các luồng mạng dương. Luồng mạng từ v_1 đến v_2 trở thành 5 – 7 = –2, và luồng mạng từ v_2 đến v_1 trở thành 7 – 5 = 2. Bởi luồng mạng từ v_2 đến v_1 là dương, nó biểu diễn một chuyến 2 thùng hàng mỗi ngày từ v_2 đến v_1. Luồng mạng từ v_1 đến v_2 là –2 thùng hàng mỗi ngày, và bởi luồng mạng không dương, nên không có các quả bóng khúc côn cầu được chuyển đi theo hướng này. Một cách khác, trong số 7 thùng hàng bảo đảm mỗi ngày từ v_2 đến v_1, ta có thể xem 5 thùng là khử chuyến hàng 5 thùng mỗi ngày từ v_1 đến v_2, để lại 2 thùng hàng làm chuyến hàng thực tế mỗi ngày từ v_2 đến v_1.



Hình 27.3 Chuyển đổi một bài toán luồng cực đại đa nguồn, đa bồn thành một bài toán có một nguồn đơn và một bồn đơn. **(a)** Một mạng luồng với năm nguồn $S = \{s_1, s_2, s_3, s_4, s_5,\}$ và ba bồn $T = \{t_1, t_2, t_3\}$. **(b)** Một mạng luồng nguồn đơn, bồn đơn tương đương. Ta bổ sung một siêu nguồn [supersource] s' và một cạnh có dung lượng vô hạn từ s' đến mỗi trong số nhiều nguồn. Ta cũng bổ sung một siêu bồn [supersink] t' và một cạnh có dung lượng vô hạn từ mỗi trong số nhiều bồn đến t'.

Các mạng có nhiều nguồn và bồn

Một bài toán luồng cực đại có thể có nhiều nguồn và bồn, thay vì chỉ có một. Ví dụ, công ty Lucky Puck có thể thực tế có một tập hợp m nhà máy $\{s_1, s_2,..., s_m\}$ và một tập hợp n nhà kho $\{t_1, t_2,..., t_n\}$, như đã nêu trong Hình 27.3(a). May thay, bài toán này không khó hơn luồng cực đại bình thường.

Ta có thể rút gọn bài toán xác định một luồng cực đại trong một mạng có nhiều nguồn và nhiều bồn thành bài toán luồng cực đại bình thường. Hình 27.3(b) nêu cách chuyển đổi mạng từ (a) thành mạng luồng bình thường với chỉ một nguồn đơn và một bồn đơn. Ta bổ sung một ***siêu nguồn*** s và bổ sung một cạnh có hướng (s, s_i) với dung lượng $c(s, s_i) = \infty$ cho mỗi $i = 1, 2,..., m$. Ta cũng tạo một ***siêu bồn*** mới t và bổ sung một cạnh có hướng (t_j, t) với dung lượng $c(t_j, t) = \infty$ cho mỗi $i = 1, 2,..., n$. Theo trực giác, bất kỳ luồng nào trong mạng của (a) tương ứng với một luồng trong mạng của (b), và ngược lại. Nguồn đơn s đơn giản

cung cấp luồng theo như mong muốn của nhiều nguồn s_i, và bồn đơn t cũng vậy, nó tiêu thụ luồng theo như mong muốn của nhiều bồn t_i. Bài tập 27.1-3 yêu cầu bạn chứng minh theo hình thức rằng hai bài toán là tương đương.

Làm việc với các luồng

Ta sẽ đề cập đến vài hàm (như f) tiếp nhận hai đỉnh trong một mạng luồng làm đối số. Trong chương này, ta sẽ dùng một ***hệ ký hiệu phép lấy tổng ẩn*** ở đó một trong hai, hoặc cả hai, đối số có thể là một *tập hợp* các đỉnh, và giá trị biểu hiện được hiểu là tổng của tất cả cách thay thế các đối số khả dĩ bằng các phần tử của chúng. Ví dụ, nếu X và Y là những tập hợp các đỉnh, thì

$$f(X, Y) = \sum_{x \in X} \sum_{y \in Y} f(x, y) \ .$$

Để lấy một ví dụ khác, ta có thể diễn tả sự ràng buộc bảo toàn luồng như là điều kiện $f(u, V) = 0$ với tất cả $u \in V - \{s, t\}$. Ngoài ra, để tiện dụng, ta thường bỏ qua các dấu ngoặc ôm tập hợp khi chúng được dùng trong hệ ký hiệu phép lấy tổng ẩn. Ví dụ, trong phương trình $f(s, V - s) = f(s, V)$, số hạng $V - s$ có nghĩa là tập hợp $V - \{s\}$.

Ký hiệu tập hợp ẩn thường đơn giản hóa các phương trình có liên quan đến các luồng. Bổ đề dưới đây, mà phần chứng minh của nó được để lại làm Bài tập 27.1-4, sẽ chốt giữ vài đồng nhất thức thường gặp nhất có liên quan đến các luồng và ký hiệu tập hợp ẩn.

Bổ đề 27.1

Cho $G = (V, E)$ là một mạng luồng, và cho f là một luồng trong G. Thì, với $X \subseteq V$,

$$f(X, X) = 0 \ .$$

Với $X, Y \subseteq V$,

$$f(X, Y) = -f(Y, X) \ .$$

Với $X, Y, Z \subseteq V$ mà $X \cap Y = \emptyset$,

$$f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$$

và

$$f(Z, X \cup Y) = f(Z, X) + f(Z, Y) \ .$$

Để lấy ví dụ về cách làm việc với hệ ký hiệu phép lấy tổng ẩn, ta có thể chứng minh rằng giá trị của một luồng là tổng luồng mạng vào bồn; nghĩa là,

$$|f| = f(V, t) \ . \qquad (27.3)$$

Theo trực giác điều này là đúng, bởi tất cả các đỉnh khác với nguồn và bồn đều có một luồng mạng 0 do sự bảo toàn luồng lưu, và như vậy bồn là đỉnh duy nhất khác có thể có một luồng mạng phi zero để so khớp luồng mạng phi zero của nguồn. Phần chứng minh hình thức của chúng ta sẽ như sau:

$$
\begin{aligned}
|f| &= f(s, V) && \text{(theo định nghĩa)} \\
&= f(V, V) - f(V - s, V) && \text{(theo Bổ đề 27.1)} \\
&= f(V, V - s) && \text{(theo Bổ đề 27.1)} \\
&= f(V, t) + f(V, V - s - t) && \text{(theo Bổ đề 27.1)} \\
&= f(V, t) && \text{(theo sự bảo toàn luồng lưu)}\ .
\end{aligned}
$$

Ở phần sau trong chương này, ta sẽ tổng quát hóa kết quả này (Bổ đề 27.5).

Bài tập

27.1-1

Cho các đỉnh u và v trong một mạng luồng, ở đó $c(u, v) = 5$ và $c(v, u) = 8$, giả sử 3 đơn vị của luồng được chuyển đi từ u đến v và 4 đơn vị được chuyển đi từ v đến u. Đâu là luồng mạng từ u đến v? Vẽ tình huống theo kiểu dáng của Hình 27.2.

27.1-2

Xác minh mỗi trong số ba tính chất luồng cho luồng f nêu trong Hình 27.1(b).

27.1-3

Mở rộng các tính chất luồng và phần định nghĩa cho bài toán đa nguồn, đa bồn. Chứng tỏ bất kỳ luồng nào trong một mạng luồng đa nguồn, đa bồn đều tương ứng với một luồng có giá trị đồng nhất trong mạng nguồn đơn, bồn đơn có được bằng cách bổ sung một siêu nguồn và một siêu bồn, và ngược lại.

27.1-4

Chứng minh Bổ đề 27.1.

27.1-5

Với mạng luồng $G = (V, E)$ và luồng f nêu trong Hình 27.1(b), hãy tìm một cặp tập hợp con $X, Y \subseteq V$ mà $f(X, Y) = -f(V - X, Y)$. Sau đó, tìm một cặp tập hợp con $X, Y \subseteq V$ mà $f(X, Y) \neq -f(V - X, Y)$.

27.1-6

Căn cứ vào một mạng luồng $G = (V, E)$, cho f_1 và f_2 là các hàm từ $V \times V$ đến **R**. ***tổng luồng*** $f_1 + f_2$ là hàm từ $V \times V$ đến **R** được định nghĩa bởi

$$(f_1 + f_2)(u, v) = f_1(u, v) + f_2(u, v) \quad (27.4)$$

với tất cả $u, v \in V$. Nếu f_1 và f_2 là các luồng trong G, tổng luồng $f_1 + f_2$ phải thỏa tính chất nào trong số ba tính chất luồng, và có thể vi phạm tính chất nào?

27.1-7

Cho f là một luồng trong một mạng, và cho α là một số thực. Tích luồng vôhướng αf là một hàm từ $V \times V$ đến **R** được định nghĩa bởi

$$(\alpha f)(u, v) = \alpha \bullet f(u, v).$$

Chứng minh các luồng trong một mạng hình thành một tập hợp lồi bằng cách chứng tỏ nếu f_1 và f_2 là các luồng, thì như vậy là một $f_1 + (1 - \alpha) f_2$ với tất cả α trong miền giá trị $0 \le \alpha \le 1$.

27.1-8

Phát biểu bài toán luồng cực đại dưới dạng một bài toán lập trình tuyến tính.

27.1-9

Mô hình mạng luồng đã giới thiệu trong đoạn này hỗ trợ luồng của một mặt hàng; một ***mạng luồng nhiều mặt hàng*** hỗ trợ luồng p mặt hàng giữa một tập hợp ***p đỉnh nguồn*** $S = \{s_1, s_2, \ldots, s_p\}$ và một tập hợp ***p đỉnh bồn*** $T = \{t_1, t_2, \ldots, t_p\}$. Luồng mạng của mặt hàng thứ i từ u đến v được ký hiệu là $f_i(u, v)$. Với mặt hàng thứ i, nguồn duy nhất là s_i và bồn duy nhất là t_i. Ta có sự bảo toàn luồng lưu độc lập cho mỗi mặt hàng: luồng mạng của mỗi mặt hàng rời mỗi đỉnh là zero trừ phi đỉnh là nguồn hoặc bồn cho mặt hàng đó. Tổng các luồng mạng của tất cả các mặt hàng từ u đến v không được vượt quá $c(u, v)$, và các luồng mặt hàng tương tác theo cách này. ***Giá trị*** của luồng mỗi mặt hàng là luồng mạng rời nguồn cho mặt hàng đó. Tổng giá trị luồng là tổng các giá trị của tất cả p luồng mặt hàng. Chứng minh có một thuật toán thời gian đa thức giải quyết bài toán tìm tổng giá trị luồng cực đại của một mạng luồng đa mặt hàng bằng cách trình bày bài toán dưới dạng một chương trình tuyến tính.

27.2 Phương pháp Ford-Fulkerson

Đoạn này trình bày phương pháp Ford-Fulkerson để giải quyết bài toán luồng cực đại. Ta gọi nó là một "phương pháp" thay vì một "thuật toán" bởi nó bao hàm vài thực thi có các thời gian thực hiện khác nhau. Phương pháp Ford-Fulkerson tùy thuộc vào ba ý tưởng quan trọng vượt trên phương pháp và có liên quan đến nhiều thuật toán và bài toán luồng: các mạng thặng dư, các lộ trình tăng cường, và các phần cắt. Các ý tưởng này là thiết yếu đối với định lý max-flow min-cut quan trọng (Định lý 27.7), định rõ đặc điểm đối với giá trị của một luồng cực đại theo dạng các phần cắt của mạng luồng. Để kết thúc đoạn này, ta trình bày một kiểu thực thi cụ thể của phương pháp Ford-Fulkerson và phân tích thời gian thực hiện của nó.

Ford-Fulkerson là phương pháp lặp. Ta bắt đầu với $f(u, v) = 0$ với tất cả $u, v \in V$, cho ra một luồng ban đầu có giá trị 0. Vào mỗi lần lặp lại, ta tăng giá trị luồng bằng cách tìm một "lộ trình tăng cường," mà ta có thể đơn giản xem nó như một lộ trình từ nguồn s đến bồn t dọc theo đó ta có thể đẩy thêm luồng, rồi tăng cường luồng dọc theo lộ trình này. Ta lặp lại tiến trình này cho đến khi không tìm thấy lộ trình tăng cường nào. Định lý max-flow min-cut sẽ chứng tỏ khi kết thúc, tiến trình này cho ra một luồng cực đại.

FORD-FULKERSON-METHOD(G, s, t)

1 khởi tạo luồng f theo 0

2 **while** ở đó tồn tại một lộ trình tăng cường p

3   **do** tăng cường luồng f dọc theo p

4 **return** f

Các mạng thặng dư

Theo trực giác, cho một mạng luồng và một luồng, mạng thặng dư bao gồm các cạnh có thể tiếp nhận thêm luồng mạng. Chính thức hơn, giả sử ta có một mạng luồng $G = (V, E)$ với nguồn s và bồn t. Cho f là một luồng trong G, và xét một cặp đỉnh $u, v \in V$. Khối lượng của luồng mạng *bổ sung* mà ta có thể đẩy từ u đến v trước khi vượt quá dung lượng $c(u, v)$ là ***dung lượng thặng dư*** của (u, v), căn cứ vào

$$c_f(u,v) = c(u,v) - f(u, v) . \tag{27.5}$$

Ví dụ, nếu $c(u, v) = 16$ và $f(u, v) = 11$, thì ta có thể chuyển đi thêm $c_f(u, v) = 5$ đơn vị của luồng trước khi vượt quá sự ràng buộc dung lượng trên cạnh (u, v).

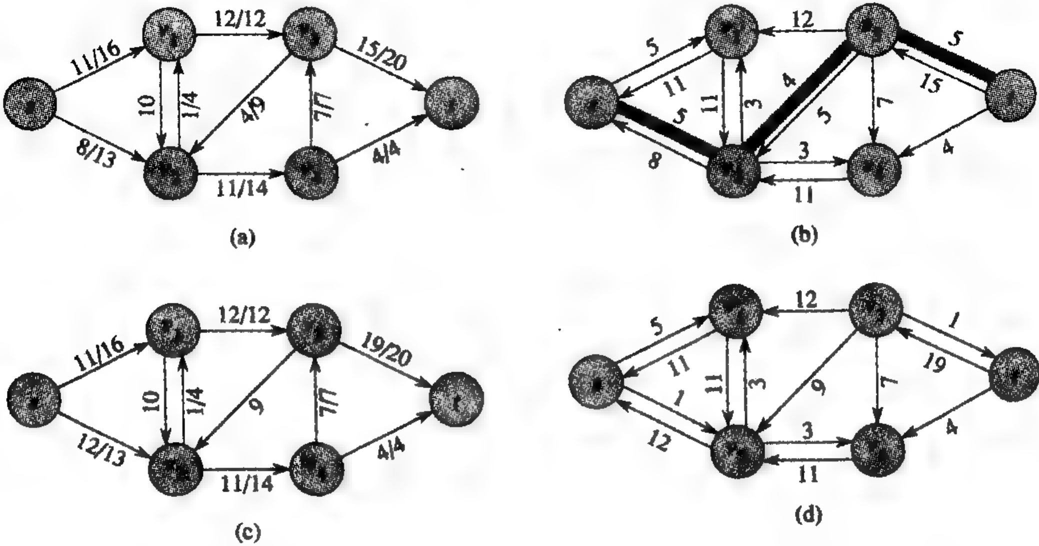
Khi luồng mạng $f(u,v)$ là âm, dung lượng thặng dư $c_f(u,v)$ sẽ lớn hơn dung lượng $c(u, v)$. Ví dụ, nếu $c(u, v) = 16$ và $f(u, v) = -4$, thì dung lượng thặng dư $c_f(u, v)$ là 20. Ta có thể diễn dịch điều này như sau. Ta có một luồng mạng 4 đơn vị từ v đến u, mà ta có thể khử bằng cách đẩy một luồng mạng 4 đơn vị từ u đến v. Sau đó, ta có thể đẩy một lượng 16 đơn vị khác từ u đến v trước khi vi phạm sự ràng buộc dung lượng trên cạnh (u, v). Như vậy, ta đã đẩy một lượng bổ sung 20 đơn vị của luồng, bắt đầu bằng một luồng mạng $f(u, v) = -4$, trước khi đụng sự ràng buộc dung lượng.

Cho một mạng luồng $G = (V, E)$ và một luồng f, ***mạng thặng dư*** của G được cảm sinh bởi f là $G_f = (V, E_f)$, ở đó

$$E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\} .$$

Nghĩa là, đã hứa hẹn trên đây, mỗi cạnh của mạng thặng dư, hoặc ***cạnh thặng dư***, có thể tiếp nhận một luồng mạng hoàn toàn dương. Hình 27.4(a) lặp lại luồng mạng G và luồng f của Hình 27.1(b), và Hình 27.4(b) nêu mạng thặng dư tương ứng G_f.

Lưu ý, (u, v) có thể là một cạnh thặng dư trong E_f cho dù nó không phải là một cạnh trong E. Nói cách khác, rất có thể là trường hợp $E_f \subsetneq E$. Mạng thặng dư trong Hình 27.4(b) bao gồm vài cạnh như vậy không



Hình 27.4 **(a)** Mạng luồng G và luồng f của Hình 27.1(b). **(b)** Mạng thặng dư G_f với lộ trình tăng cường p được tô bóng; dung lượng thặng dư của nó là $c_f(p) = c(v_2, v_3) = 4$. **(c)** Luồng trong G là kết quả từ sự tăng cường dọc theo lộ trình p bởi dung lượng thặng dư 4 của nó. **(d)** Mạng thặng dư được cảm sinh bởi luồng trong (c).

nằm trong mạng luồng ban đầu, như (v_1, s) và (v_2, v_3). Một cạnh (u, v) như vậy chỉ xuất hiện trong G_f nếu $(v, u) \in E$ và có luồng mạng dương từ v đến u. Bởi luồng mạng $f(u, v)$ từ u đến v là âm, nên $c_f(u, v) = c(u, v) - f(u, v)$ là dương và $(u, v) \in E_f$. Bởi một cạnh (u, v) chỉ có thể xuất hiện trong một mạng thặng dư nếu ít nhất một trong số (u, v) và (v, u) xuất hiện trong mạng ban đầu, ta có cận

$$|E_f| \leq 2|E|$$

Nhận thấy mạng thặng dư G_f chính là một mạng luồng có các dung lượng căn cứ vào c_f. Bổ đề dưới đây nêu rõ một luồng trong một mạng thặng dư có liên quan với một luồng trong mạng luồng ban đầu như thế nào.

Bổ đề 27.2

Cho $G = (V, E)$ là một mạng luồng với nguồn s và bồn t, và cho f là một luồng trong G. Cho G_f là mạng thặng dư của G được cảm sinh bởi f, và cho f' là một luồng trong G_f. Như vậy, tổng luồng $f + f'$ được định nghĩa bởi phương trình (27.4) chính là một luồng trong G với giá trị $|f + f'| = |f| + |f'|$.

Chứng minh Ta phải xác minh tính đối xứng ghềnh, các hạn chế dung lượng, và sự bảo toàn luồng lưu tất cả đều được tuân thủ. Với tính đối xứng ghềnh, lưu ý, với tất cả $u, v \in V$, ta có

$$\begin{aligned}(f + f')(u, v) &= f(u, v) + f'(u, v)\\ &= -f(v, u) - f'(v, u)\\ &= -(f(v, u) + f'(v, u))\\ &= -(f + f')(v, u).\end{aligned}$$

Với các hạn chế dung lượng, lưu ý $f'(u, v) \leq c_f(u, v)$ với tất cả $u, v \in V$.

Do đó, theo phương trình (27.5),

$$\begin{aligned}(f + f')(u, v) &= f(u, v) + f'(u, v)\\ &\leq f(u,v) + (c(u,v) - f(u,v))\\ &= c(u, v).\end{aligned}$$

Với sự bảo toàn luồng lưu, lưu ý, với tất cả $u \in V - \{s, t\}$, ta có

$$\begin{aligned}\sum_{v\in V} (f + f')(u, v) &= \sum_{v\in V} (f(u, v) + f'(u, v))\\ &= \sum_{v\in V} f(u, v) + \sum_{v\in V} f'(u, v)\\ &= 0 + 0\\ &= 0.\end{aligned}$$

Cuối cùng, ta có

$$\begin{aligned} |f + f'| &= \sum_{v\in V} (f + f')(s, v) \\ &= \sum_{v\in V} f(s, v) + f'(s, v) \\ &= \sum_{v\in V} f(s, v) + \sum_{v\in V} f'(s, v) \\ &= |f| + |f'| \ . \end{aligned}$$

Tăng cường các lộ trình

Cho một mạng luồng $G = (V, E)$ và một luồng f, một ***lộ trình tăng cường*** p là một lộ trình đơn giản từ s đến t trong mạng thặng dư G_f. Theo định nghĩa về mạng thặng dư, mỗi cạnh (u, v) trên một lộ trình tăng cường tiếp nhận vài luồng mạng dương bổ sung từ u đến v mà không vi phạm sự ràng buộc dung lượng trên cạnh.

Lộ trình tô bóng trong Hình 27.4(b) là một lộ trình tăng cường. Nếu xem mạng thặng dư G_f trong hình như một mạng luồng, ta có thể chuyển đi tới 4 đơn vị của luồng mạng bổ sung qua mỗi cạnh của lộ trình này mà không vi phạm sự ràng buộc dung lượng, bởi dung lượng thặng dư nhỏ nhất trên lộ trình này là $c_f(v_2, v_3) = 4$. Ta gọi lượng luồng mạng cực đại mà ta có thể chuyển đi dọc theo các cạnh của một lộ trình tăng cường p là ***dung lượng thặng dư*** của p, căn cứ vào

$$c_f(p) = \min\{c_f(u,v) : (u, v) \text{ nằm trên } p\} \ .$$

Bổ đề dưới đây, mà phần chứng minh được chừa lại cho Bài tập 27.2-7, khiến đối số trên đây chính xác hơn.

Bổ đề 27.3

Cho $G = (V, E)$ là một mạng luồng, cho f là một luồng trong G, và cho p là một lộ trình tăng cường trong G_f. Định nghĩa một hàm $f_p : V \times V \to \mathbf{R}$ với

$$f_p(u, v) = \begin{cases} c_f(p) \text{ nếu } (u, v) & \text{nằm trên } p \ , \\ -c_f(p) \text{ nếu } (v, u) & \text{nằm trên } p \ , \\ 0 & \text{bằng không.} \end{cases} \qquad (27.6)$$

Như vậy, f_p là một luồng trong G_f với giá trị $|f_p| = c_f(p) > 0$.

Hệ luận dưới đây chứng tỏ nếu bổ sung f_p vào f, ta có một luồng khác trong G mà giá trị của nó sát với cực đại hơn. Hình 27.4(c) nêu kết quả của việc bổ sung f_p trong Hình 27.4(b) vào f của Hình 27.4(a).

Hệ luận 27.4

Cho $G = (V, E)$ là một mạng luồng, cho f là một luồng trong G, và cho

p là một lộ trình tăng cường trong G_f. Cho f_p được định nghĩa như trong phương trình (27.6). Định nghĩa một hàm $f': V \times V \to \mathbf{R}$ với $f' = f + f_p$. Như vậy, f là một luồng trong G có giá trị $|f'| = |f| + |f_p| > |f|$.

Chứng minh Tức thời từ các Bổ đề 27.2 và 27.3.

Các phần cắt của các mạng luồng

Phương pháp Ford-Fulkerson liên tục tăng cường luồng dọc theo các lộ trình tăng cường cho đến khi tìm thấy một luồng cực đại. Định lý max-flow min-cut, mà ta sẽ chứng minh dưới đây, cho biết một luồng là cực đại nếu và chỉ nếu mạng thặng dư của nó không chứa lộ trình tăng cường. Tuy nhiên, để chứng minh định lý này, trước tiên ta phải khảo sát khái niệm phần cắt của một mạng luồng.

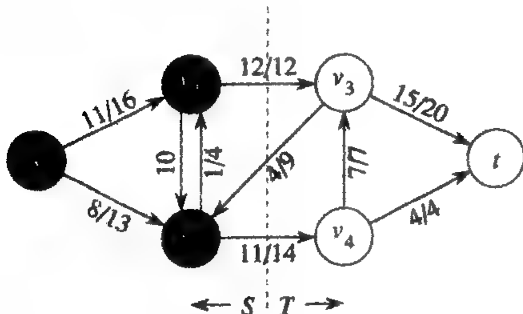
Một ***phần cắt*** (S, T) của mạng luồng $G = (V, E)$ là một phân hoạch của V vào S và $T = V - S$ sao cho $s \in S$ và $t \in T$. (Định nghĩa này cũng giống như định nghĩa "phần cắt" mà ta đã dùng cho các cây tỏa nhánh cực tiểu trong Chương 24, ngoại trừ, ở đây ta đang cắt một đồ thị có hướng thay vì một đồ thị không hướng, và nhấn mạnh rằng $s \in S$ và $t \in T$.) Nếu f là một luồng, thì ***luồng mạng*** qua phần cắt (S, T) được định nghĩa là $f(S, T)$. ***Dung lượng*** của phần cắt (S, T) là $c(S, T)$.

Hình 27.5 nêu phần cắt $(\{s, v_1, v_2\}, \{v_3, v_4, t\})$ trong mạng luồng của Hình 27.1(**b**). Luồng mạng qua phần cắt này là

$$\begin{aligned} f(v_1, v_3) + f(v_2, v_3) + f(v_2, v_4) &= 12 + (-4) + 11 \\ &= 19, \end{aligned}$$

và dung lượng của nó là

$$\begin{aligned} c(v_1, v_3) + c(v_2, v_4) &= 12 + 14 \\ &= 26 . \end{aligned}$$



Hình 27.5 Một phần cắt (S, T) trong mạng luồng của Hình 27.1(**b**), ở đó $S = \{s, v_1, v_2\}$ và $T = \{v_3, v_4, t\}$. Các đỉnh trong S mang màu đen, và các đỉnh trong T mang màu trắng. Luồng mạng qua (S, T) là $f(S, T) = 19$, và dung lượng là $c(S, T) = 26$.

Nhận thấy luồng mạng qua một phần cắt có thể gộp các luồng mạng âm giữa các đỉnh, nhưng dung lượng của một phần cắt hoàn toàn được hình thành bởi các giá trị không âm.

Bổ đề dưới đây chứng tỏ giá trị của một luồng trong một mạng là luồng mạng qua bất kỳ phần cắt nào của mạng.

Bổ đề 27.5

Cho f là một luồng trong một mạng luồng G với nguồn s và bồn t, và cho (S, T) là một phần cắt của G. Như vậy, luồng mạng qua (S, T) là $f(S, T) = |f|$.

Chứng minh Dùng Bổ đề 27.1, ta có

$$\begin{aligned} f(S, T) &= f(S, V) - f(S, S) \\ &= f(S, V) \\ &= f(s, V) + f(S - s, V) \\ &= f(s, V) \\ &= |f| . \end{aligned}$$

Một hệ luận tức thời đối với Bổ đề 27.5 đó là kết quả mà ta đã chứng minh trên đây—phương trình (27.3)—rằng giá trị của một luồng chính là luồng mạng vào bồn.

Một hệ luận khác cho Bổ đề 27.5 nêu cách dùng các dung lượng cắt để định cận giá trị của một luồng.

Hệ luận 27.6

Giá trị của bất kỳ luồng f nào trong một mạng luồng G đều được định cận từ bên trên bằng dung lượng của bất kỳ phần cắt nào của G.

Chứng minh Cho (S, T) là một phần cắt bất kỳ của G và cho f là một luồng bất kỳ. Theo Bổ đề 27.5 và các hạn chế dung lượng,

$$\begin{aligned} |f| &= f(S, T) \\ &= \sum_{v \in T} \sum_{u \in S} f(u, v) \\ &\le \sum_{u \in S} \sum_{v \in T} c(u, v) \\ &= c(S, T) . \end{aligned}$$

Giờ đây ta sẵn sàng chứng minh định lý max-flow min-cut quan trọng.

Định lý 27.7 (Định lý max-flow min-cut)

Nếu f là một luồng trong một mạng luồng $G = (V, E)$ với nguồn s và bồn t, thì các điều kiện dưới đây là tương đương:

1. f là một luồng cực đại trong G.

2. Mạng thặng dư G_f không chứa các lộ trình tăng cường.

3. $|f| = c(S, T)$ với vài phần cắt (S, T) của G.

Chứng minh (1) ⇒ (2): Vì sự mâu thuẫn, ta giả sử f là một luồng cực đại trong G nhưng G_f có một lộ trình tăng cường p. Thì, theo Hệ luận 27.4, tổng luồng $f + f_p$, ở đó f_p căn cứ vào phương trình (27.6), là một luồng trong G với giá trị hoàn toàn lớn hơn $|f|$, mâu thuẫn với giả thiết rằng f là một luồng cực đại.

(2) ⇒ (3): Giả sử rằng G_f không có lộ trình tăng cường nào, nghĩa là, G_f không chứa lộ trình từ s đến t. Định nghĩa

$S = \{v \in V$: ở đó tồn tại một lộ trình từ s đến v trong $G_f\}$

và $T = V - S$. Phân hoạch (S, T) là một phần cắt: ta có $s \in S$ theo lẽ thường và $t \notin S$ bởi có không lộ trình từ s đến T trong G_f. Với mỗi cặp đỉnh u và v sao cho $u \in S$ và $v \in T$, ta có $f(u,v) = c(u,v)$, bởi bằng không $(u, v) \in E_f$ và v nằm trong tập hợp S. Theo Bổ đề 27.5, do đó,

$|f| = f(S, T) = c(S, T)$.

(3) ⇒ (1): Theo Hệ luận 27.6, $|f| \le c(S, T)$ với tất cả các phần cắt (S, T). Như vậy, điều kiện $|f| = c(S, T)$ hàm ý rằng f là một luồng cực đại.

Thuật toán Ford-Fulkerson căn bản

Trong mỗi lần lặp lại của phương pháp Ford-Fulkerson, ta tìm *bất kỳ* lộ trình tăng cường p nào và tăng cường luồng f dọc theo p theo dung lượng thặng dư $c_f(p)$. Kiểu thực thi dưới đây của phương pháp tính toán luồng cực đại trong một đồ thị $G = (V, E)$ bằng cách cập nhật luồng mạng $f[u, v]$ giữa mỗi cặp u, v của các đỉnh được nối bằng một cạnh.[1] Nếu *u và v* không được nối bằng một cạnh theo một trong hai hướng, ta mặc nhận rằng $f(u, v] = 0$. Mã mặc nhận dung lượng từ u đến v được cung cấp bởi một một hàm thời gian bất biến $c(u, v)$, với $c(u, v) = 0$ nếu $(u, v) \notin E$. (Trong một kiểu thực thi điển hình, $c(u, v)$ có thể phái sinh từ các trường được lưu trữ trong các đỉnh và các danh sách kề của chúng.) Dung lượng thặng dư $c_f(u, v)$ được tính toán theo công thức (27.5). Biểu thức $c_f(p)$ trong mã thực chỉ là một biến tạm thời lưu trữ dung lượng thặng dư của lộ trình p.

FORD-FULKERSON(G, s, t)

---

[1] Ta dùng các dấu ngoặc vuông khi xem một dấu định danh—chẳng hạn f—như một trường khả biến, và ta dùng các dấu ngoặc đơn khi xem nó như một hàm.

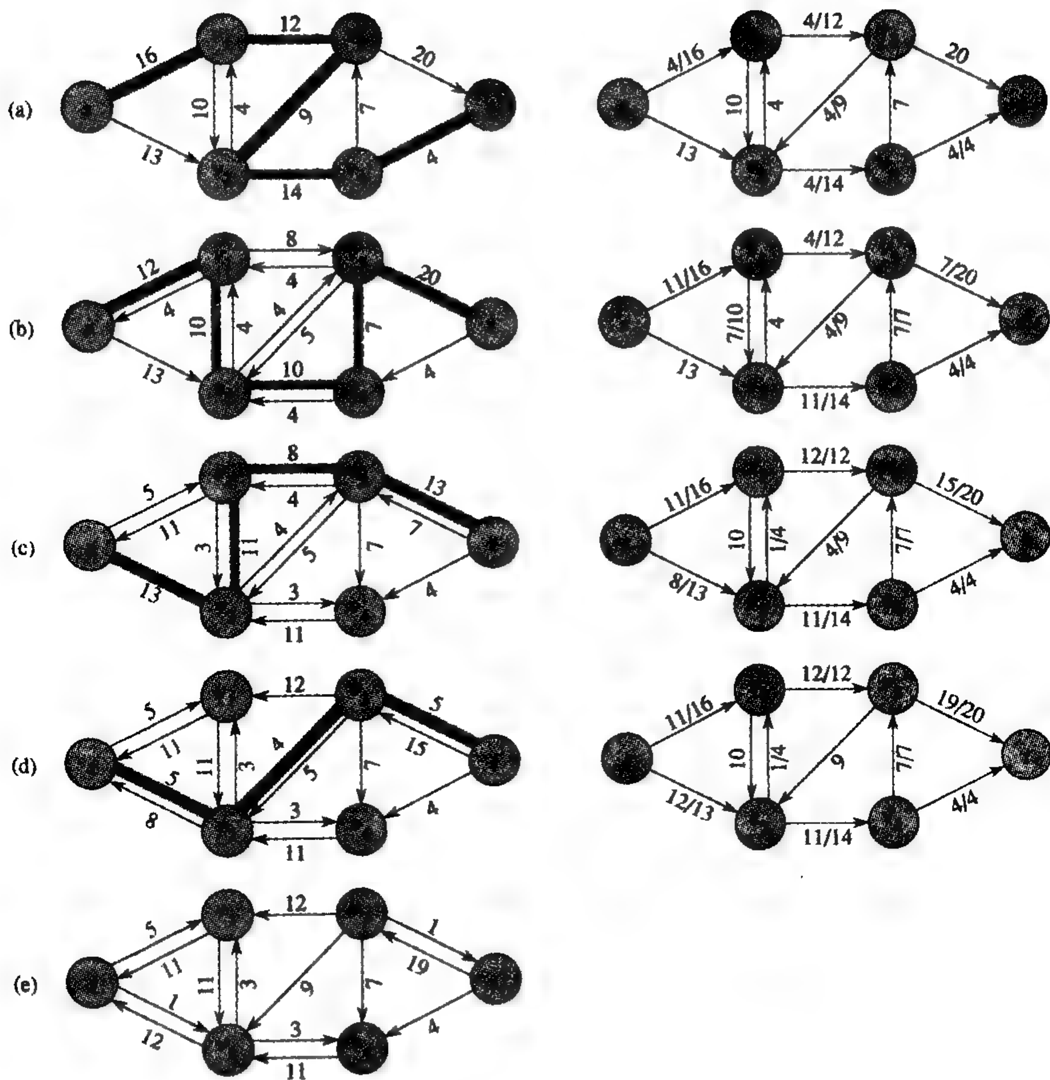
```
1 for mỗi cạnh (u, v) ∈ E[G]
2     do f[u, v] ← 0
3        f[v, u] ← 0
4 while ở đó tồn tại một lộ trình p từ s đến t trong mạng thặng dư G_f
5     do c_f(p) ← min {c_f(u, v) : (u, v) nằm trong p}
6        for mỗi cạnh (u, v) trong p
7            do f[u,v] ← f[u,v] + c_f(p)
8               f[v, u] ← -f[u, v]
```

Thuật toán FORD-FULKERSON đơn giản mở rộng trên mã giả FORD-FULKERSON-METHOD đã cho trên đây. Hình 27.6 nêu kết quả của mỗi lần lặp lại trong một đợt chạy mẫu. Các dòng 1-3 khởi tạo luồng f theo 0. Vòng lặp **while** của các dòng 4-8 liên tục tìm một lộ trình tăng cường p trong G_f và tăng cường luồng f dọc theo p theo dung lượng thặng dư $c_f(p)$. Khi không có lộ trình tăng cường nào tồn tại, luồng f là một luồng cực đại.

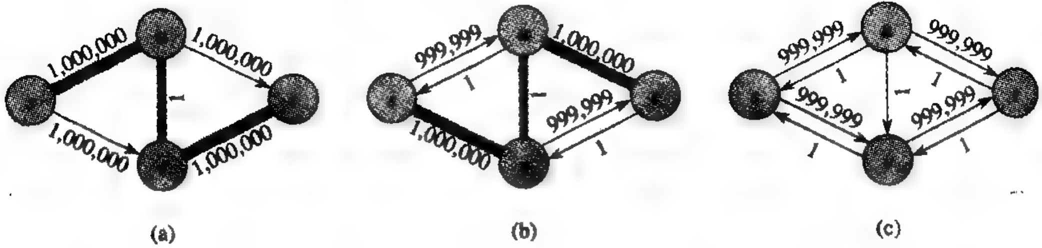
Phân tích Ford-Fulkerson

Thời gian thực hiện của FORD-FULKERSON tùy thuộc vào cách xác định lộ trình tăng cường p trong dòng 4. Nếu nó được chọn tồi, thuật toán thậm chí có thể không kết thúc: giá trị của luồng sẽ gia tăng cùng với các lần tăng cường liên tục, nhưng thậm chí nó không cần hội tụ theo giá trị cực đại luồng. Tuy nhiên, nếu lộ trình tăng cường được chọn bằng cách dùng một thuật toán tìm kiếm độ rộng đầu tiên (Đoạn 23.2), thuật toán chạy trong thời gian đa thức. Tuy nhiên, trước khi chứng minh điều này, ta được một cận đơn giản cho trường hợp ở đó lộ trình tăng cường được chọn một cách tùy ý và tất cả các dung lượng là số nguyên.

Đa phần trong thực tế, bài toán luồng cực đại nảy sinh với các dung lượng tích phân. Nếu các dung lượng là số hữu tỷ, ta có thể dùng một phép biến đổi định tỷ lệ thích hợp để chuyển tất cả thành tích phân. Dưới giả thiết này, một thực thi đơn giản của FORD-FULKERSON sẽ chạy trong thời gian $O(E|f^*|)$, ở đó f^* là luồng cực đại mà thuật toán tìm thấy. Phần phân tích sẽ như sau. Các dòng 1-3 chiếm thời gian $\Theta(E)$. Vòng lặp **while** của các dòng 4-8 được thi hành tối đa $|f^*|$ lần, bởi giá trị luồng gia tăng ít nhất theo một đơn vị trong mỗi lần lặp lại.



Hình 27.6 Thi hành thuật toán Ford-Fulkerson. **(a)-(d)** Các lần lặp lại liên tục của vòng lặp **while**. Cạnh trái của từng phần sẽ nêu mạng thặng dư G_f từ dòng 4 bằng một lộ trình tăng cường p được tô bóng. Cạnh phải của từng phần sẽ nêu luồng mới f kết quả của việc bổ sung f_p vào f. Mạng thặng dư trong (a) là nhập liệu mạng G. **(e)** Mạng thặng dư tại lần trắc nghiệm vòng lặp **while** . Nó không có các lộ trình tăng cường, và do đó luồng f nêu trong (d) là một luồng cực đại.



Hình 27.7 (a) Một mạng luồng mà FORD-FULKERSOIV có thể mất $\Theta(E|f^*|)$ thời gian, ở đó f^* là một luồng cực đại, được nêu ở đây bằng $|f^*| =$ 2,000,000. Một lộ trình tăng cường với dung lượng thặng dư 1 được nêu. **(b)** Mạng thặng dư kết quả. Một lộ trình tăng cường khác với dung lượng thặng dư 1 được nêu. **(c)** Mạng thặng dư kết quả.

Công việc thực hiện trong vòng lặp **while** có thể sẽ hiệu quả nếu ta quản lý một cách hiệu quả cấu trúc dữ liệu được dùng để thực thi mạng $G = (V, E)$. Hãy mặc nhận rằng ta duy trì một cấu trúc dữ liệu tương ứng với một đồ thị có hướng $G' = (V, E')$, ở đó $E' = \{(u,v) : (u, v) \in E$ hoặc $(v, u) \in E\}$. Các cạnh trong mạng G cũng là các cạnh trong G', và do đó ta chỉ cần duy trì các dung lượng và các luồng trong cấu trúc dữ liệu này. Cho một luồng f trên G, các cạnh trong mạng thặng dư G_f bao gồm tất cả các cạnh (u, v) của G' sao cho $c(u, v) - f[u, v] \neq 0$. Do đó, thời gian để tìm một lộ trình trong một mạng thặng dư là $O(E') = O(E)$ nếu ta dùng thuật toán tìm kiếm độ sâu đầu tiên hoặc tìm kiếm độ rộng đầu tiên. Như vậy, mỗi lần lặp lại của vòng lặp **while** sẽ mất $O(E)$ thời gian, khiến tổng thời gian thực hiện của FORD-FULKERSON trở thành $O(E|f^*|)$.

Khi các dung lượng là tích phân và giá trị luồng tối ưu $|f^*|$ nhỏ, thời gian thực hiện của thuật toán Ford-Fulkerson tỏ ra thích hợp. Hình 27.7(a) có nêu một ví dụ về nội dung có thể xảy ra trên một mạng luồng đơn giản mà $|f^*|$ là lớn. Một luồng cực đại trong mạng này có giá trị 2,000,000: 1,000,000 đơn vị của luồng băng ngang lộ trình $s \to u \to t$, và một 1,000,000 đơn vị khác băng ngang lộ trình $s \to v \to t$. Nếu lộ trình tăng cường đầu tiên mà FORD-FULKERSON tìm thấy là $s \to u \to v \to t$, xem Hình 27.7(a), luồng có giá trị 1 sau lần lặp lại đầu tiên. Mạng thặng dư kết quả được nêu trong Hình 27.7(b). Nếu lần lặp lại thứ hai tìm thấy lộ trình tăng cường $s \to v \to u \to t$, như đã nêu trong Hình 27.7(b), thì luồng sẽ giá trị 2. Hình 27.7(c) nêu mạng thặng dư kết quả. Ta có thể tiếp tục chọn lộ trình tăng cường $s \to u \to v \to t$ trong các lần lặp lại đánh số lẻ và lộ trình tăng cường $s \to v \to u \to t$ trong các lần lặp

lại đánh số chẵn. Ta sẽ thực hiện một tổng 2,000,000 lần tăng cường, mỗi lần chỉ tăng giá trị luồng lên 1 đơn vị.

Có thể cải thiện cận trên FORD-FULKERSON nếu ta thực thi phép tính của lộ trình tăng cường p trong dòng 4 bằng một thuật toán tìm kiếm độ rộng đầu tiên, nghĩa là, nếu lộ trình tăng cường là một lộ trình *ngắn nhất* từ s đến t trong mạng thặng dư, ở đó mỗi cạnh có khoảng cách đơn vị (trọng số). Ta gọi phương pháp Ford-Fulkerson đó là đã thực thi *thuật toán Edmonds-Karp*. Giờ đây ta chứng minh thuật toán Edmonds-Karp chạy trong $O(VE^2)$ thời gian.

Phần phân tích tùy thuộc vào các khoảng cách đến các đỉnh trong mạng thặng dư G_f. Bổ đề dưới đây sử dụng hệ ký hiệu $\delta_f(u, v)$ với khoảng cách lộ trình ngắn nhất từ u đến v trong G_f, ở đó mỗi cạnh có khoảng cách đơn vị.

Bổ đề 27.8

Nếu thuật toán Edmonds-Karp chạy trên một mạng luồng $G = (V, E)$ với nguồn s và bồn t, thì với tất cả các đỉnh $v \in V - \{s, t\}$, khoảng cách lộ trình ngắn nhất $\delta_f(s, v)$ trong mạng thặng dư G_f sẽ gia tăng đơn điệu với mỗi phép tăng cường luồng.

Chứng minh Vì sự mâu thuẫn, ta giả sử với một đỉnh $v \in V - \{s, t\}$, ta có một phép tăng cường luồng khiến $\delta_f(s, v)$ giảm. Cho f là luồng ngay trước phép tăng cường, và cho f' là luồng ngay sau đó. Thì,

$$\delta_{f'}(s,v) < \delta_f(s, v).$$

Ta có thể mặc nhận mà không làm mất tính tổng quát rằng $\delta_{f'}(s, v) \leq (\delta_{f'}(s, u)$ với tất cả các đỉnh $u \in V - \{s, t\}$ sao cho $\delta_{f'}(s, u) < \delta_f(s, u)$. Tương đương, ta có thể mặc nhận rằng với tất cả các đỉnh $u \in V - \{s, t\}$,

$$\delta_{f'}(s, u) < \delta_{f'}(s, v) \text{ hàm ý } \delta_f(s, u) \leq \delta_{f'}(s, u). \qquad (27.7)$$

Giờ đây ta lấy một lộ trình ngắn nhất p' trong $G_{f'}$, có dạng $s \leadsto u \rightarrow v$ và xét đỉnh u đứng trước v trên lộ trình này. Ta phải có $\delta_{f'}(s, u) = \delta_{f'}(s, v) - 1$ theo Hệ luận 25.2, bởi (u, v) là một cạnh trên p', chính là một lộ trình ngắn nhất từ s đến v. Do đó, theo giả thiết (27.7) của chúng ta,

$$\delta_f(s, u) \leq \delta_{f'}(s, u).$$

Như vậy, với các đỉnh v và u được thiết lập, ta có thể xét luồng mạng f từ u đến v trước khi phép tăng cường của luồng trong G_f. Nếu $f[u, v] < c(u, v)$, thì ta có

$$\begin{aligned}
\delta_f(s, v) &\leq \delta_f(s, u) + 1 \text{ (theo Bổ đề 25.3)} \\
&\leq \delta_{f'}(s, u) + 1 \\
&= \delta_{f'}(s,v),
\end{aligned}$$

mâu thuẫn với giả thiết của chúng ta rằng phép tăng cường luồng giảm khoảng cách từ s đến v.

Như vậy, ta phải có $f[u, v] = c(u, v)$, có nghĩa là $(u, v) \notin E_f$. Giờ đây, lộ trình tăng cường p đã được chọn trong G_f để tạo ra $G_{f'}$, phải chứa cạnh (v, u) theo hướng từ v đến u, bởi $(u, v) \in E_{f'}$, (theo giả thuyết) và $(u, v) \notin E_f$ như ta vừa nêu. Nghĩa là, việc tăng cường luồng dọc theo lộ trình p đẩy luồng *trở lại* dọc theo (u, v), và v xuất hiện trước u trên p. Bởi p là một lộ trình ngắn nhất từ s đến t, các lộ trình con của nó là các lộ trình ngắn nhất (Bổ đề 25.1), và như vậy ta có $\delta_f(s, u) = \delta_f(s, v) + 1$. Bởi vậy,

$$\begin{aligned} \delta_{f'}(s, v) &= \delta_{f'}(s, u) - 1 \\ &\leq \delta_f(s, u) - 1 \\ &= \delta_f(s, v) - 2 \\ &< \delta_f(s, v), \end{aligned}$$

mâu thuẫn với giả thiết ban đầu của chúng ta.

Định lý dưới đây định cận số lần lặp lại của thuật toán Edmonds-Karp.

Định lý 27.9

Nếu thuật toán Edmonds-Karp chạy trên một mạng luồng $G = (V, E)$ với nguồn s và bồn t, thì tổng số lần tăng cường luồng mà thuật toán thực hiện sẽ tối đa là $O(VE)$.

Chứng minh Ta nói rằng một cạnh (u, v) trong một mạng thặng dư G_f là ***tới hạn*** trên một lộ trình tăng cường p nếu dung lượng thặng dư của p là dung lượng thặng dư của (u, v), nghĩa là, nếu $c_f(p) = c_f(u, v)$. Sau khi đã tăng cường luồng dọc theo một lộ trình tăng cường, mọi cạnh tới hạn trên lộ trình đều sẽ biến mất khỏi mạng thặng dư. Hơn nữa, ít nhất một cạnh trên bất kỳ lộ trình tăng cường nào đều phải tới hạn.

Cho u *và* v là các đỉnh trong V được liên thông bởi một cạnh trong E. (u, v) có thể là một cạnh tới hạn bao nhiêu lần trong khi thi hành thuật toán Edmonds-Karp? Bởi các lộ trình tăng cường là các lộ trình ngắn nhất, khi (u, v) tới hạn lần đầu tiên, nên ta có

$$\delta_f(s, v) = \delta_f(s, u) + 1 .$$

Một khi luồng được tăng cường, cạnh (u, v) biến mất khỏi mạng thặng dư. Nó không thể tái xuất hiện về sau trên một lộ trình tăng cường khá cho đến sau khi luồng mạng từ u đến v giảm, và điều này chỉ xảy ra nếu (v, u) xuất hiện trên một lộ trình tăng cường. Nếu f' là luồng trong G khi sự kiện này xảy ra, thì ta có

$\delta_{f'}(s, u) = \delta_{f'}(s, v) + 1$.

Bởi $\delta_f(s, v) \leq \delta_{f'}(s, v)$ theo Bổ đề 27.8, ta có

$$\begin{aligned}\delta_{f'}(s,u) &= \delta_{f'}(s, v) + 1 \\ &\geq \delta_f(s,v)+ 1 \\ &= \delta_f(s, u) + 2 .\end{aligned}$$

Kết quả là, từ lúc mà (u, v) trở thành tới hạn đến lúc nó trở thành tới hạn sau đó, khoảng cách của u từ nguồn sẽ gia tăng ít nhất là 2. Thoạt đầu khoảng cách của u từ nguồn ít nhất là 1, và cho đến khi nó trở thành bất khả dụng từ nguồn, nếu như có, thì khoảng cách củanó tối đa là $|V| - 2$. Như vậy, (u, v) có thể trở thành tới hạn tối đa $O(V)$ lần. Bởi có $O(E)$ cặp đỉnh có thể có một cạnh giữa chúng trong một đồ thị thặng dư, nên tổng các cạnh tới hạn trong nguyên cả tiến trình thi hành thuật toán Edmonds-Karp là $O(VE)$. Mỗi lộ trình tăng cường có ít nhất một cạnh tới hạn, và do đó định lý đứng vững.

Bởi mỗi lần lặp lại của FORD-FULKERSON có thể được thực thi trong $O(E)$ thời gian khi thuật toán tìm kiếm độ rộng đầu tiên tìm thấy lộ trình tăng cường, nên tổng thời gian thực hiện của thuật toán Edmonds-Karp là $O(VE^2)$. Thuật toán trong Đoạn 27.4 cho ta một phương pháp để đạt được một thời gian thực hiện $O(V^2E)$, lập thành cơ sở cho thuật toán $O(V^3)$ thời gian của Đoạn 27.5.

Bài tập

27.2-1

Trong Hình 27.1(b), nêu luồng qua phần cắt $(\{s, v_2, v_4\}, \{v_1, v_3, t\})$? Đâu là dung lượng của phần cắt này?

27.2-2

Nêu tiến trình thi hành thuật toán Edmonds-Karp trên mạng luồng của Hình 27.1(a).

27.2-3

Trong ví dụ của Hình 27.6, nêu phần cắt cực tiểu tương ứng với luồng cực đại đã nêu? Trong số các lộ trình tăng cường xuất hiện trong ví dụ, nêu hai lộ trình khử luồng đã được chuyển đi trước đó?

27.2-4

Chứng minh với bất kỳ cặp đỉnh u và v và bất kỳ dung lượng và hàm luồng c và f, ta có $c_f(u, v) + c_f(v, u) = c(u, v) + c(v, u)$.

27.2-5

Hãy nhớ lại phần xây dựng trong Đoạn 27.1 chuyển đổi một mạng luồng đa nguồn, đa luồng thành một mạng nguồnđơn, bồn đơn bổ sung các cạnh có dung lượng vô hạn. Chứng minh rằng bất kỳ luồng nào trong mạng kết quả đều có một giá trị hữu hạn nếu các cạnh của mạng đa nguồn, đa luồng ban đầu có dung lượng hữu hạn.

27.2-6

Giả sử mỗi nguồn s_i trong một bài toán đa nguồn, đa luồng tạo ra chính xác p_i đơn vị của luồng, sao cho $f(s_i, V) = p_i$. Cũng giả sử mỗi bồn t_j tiêu thụ chính xác q_j đơn vị, sao cho $f(V, t_j) = q_j$, ở đó $\Sigma_i p_i = \Sigma_j q_j$. Nêu cách chuyển đổi bài toán tìm một luồng f tuân thủ các hạn chế bổ sung này thành bài toán tìm một luồng cực đại trong một mạng luồng nguồn đơn, bồn đơn.

27.2-7

Chứng minh Bổ đề 27.3.

27.2-8

Chứng tỏ một luồng cực đại trong một mạng $G = (V, E)$ luôn có thể tìm thấy bởi một dãy tối đa $|E|$ lộ trình tăng cường. (*Mách nước:* Xác định các lộ trình *sau khi* tìm luồng cực đại.)

27.2-9

Khả năng liên thông cạnh [edge connectivity] của một đồ thị không hướng là số lượng cực tiểu k cạnh phải được gỡ bỏ để làm gián đoạn đồ thị. Ví dụ, khả năng liên thông cạnh của một cây là 1, và khả năng liên thông cạnh của một xích chu trình các đỉnh là 2. Nêu cách xác định khả năng liên thông cạnh của một đồ thị không hướng $G = (V, E)$ bằng cách chạy một thuật toán luồng cực đại trên tối đa $|V|$ mạng luồng, mỗi mạng có $O(V)$ đỉnh và $O(E)$ cạnh.

27.2-10

Chứng tỏ thuật toán Edmonds-Karp kết thúc sau tối đa $|V| |E| /4$ lần lặp lại. (*Mách nước.* Với một cạnh (u, v), xét cách thay đổi của cả $\delta(s, u)$ lẫn $\delta(u, t)$ giữa các lần ở đó (u, v) là tới hạn.)

27.3 So khớp hai nhánh cực đại

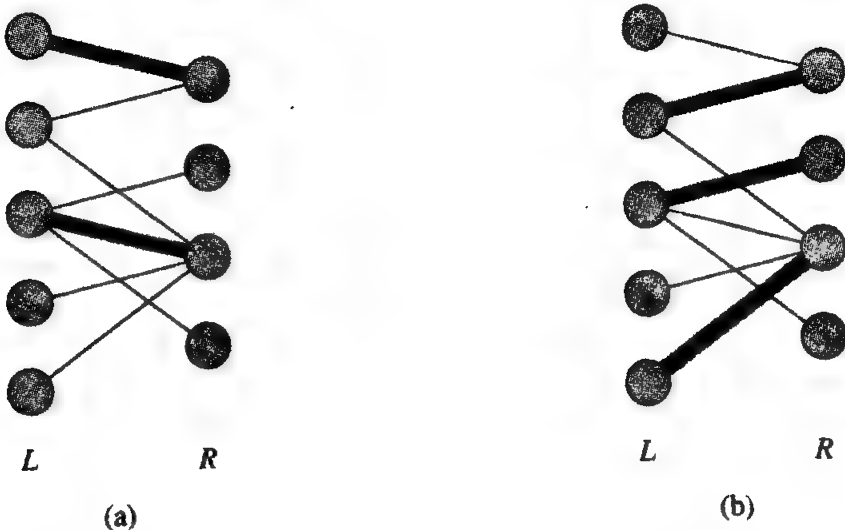
Có thể dễ dàng áp đổi vài bài toán tổ hợp dưới dạng bài toán luồng cực đại. Bài toán luồng cực đại đa nguồn, đa bồn trong Đoạn 27.1 đã

cho ta một ví dụ. Có các bài toán tổ hợp khác bề ngoài có vẻ như chẳng dính dáng gì mấy đến các mạng luồng, song thực tế có thể rút gọn thành một bài toán luồng cực đại. Đoạn này trình bày một bài toán như vậy: tìm một so khớp cực đại trong một đồ thị hai nhánh (xem Đoạn 5.4). Để giải quyết bài toán này, ta sẽ vận dụng một tính chất của tính tích phân mà phương pháp Ford-Fulkerson cung cấp. Ta cũng sẽ thấy phương pháp Ford-Fulkerson có thể được thực hiện để giải quyết bài toán so khớp hai nhánh cực đại trên một đồ thị $G = (V, E)$ trong $O(VE)$ thời gian.

Bài toán so khớp hai nhánh cực đại

Cho một đồ thị không hướng $G = (V, E)$, một ***so khớp*** [matching] là một tập hợp con các cạnh $M \subseteq E$ sao cho với tất cả các đỉnh $v \in V$, có tối đa một cạnh của M là liên thuộc trên v. Ta nói rằng một đỉnh $v \in V$ ***được so khớp*** bằng cách so khớp M nếu có một cạnh f trong M là liên thuộc trên v; bằng không, v ***không khớp*** [unmatched]. Một ***so khớp cực đại*** [maximum matching] là một so khớp bản số cực đại, nghĩa là, một so khớp M sao cho với một so khớp M' bất kỳ, ta có $|M| \geq |M'|$. Trong đoạn này, ta sẽ hạn chế sự chú ý để tìm các lần so khớp cực đại trong hai nhánh đồ thị. Ta mặc nhận tập hợp đỉnh có thể được phân hoạch thành $V = L \cup R$, ở đó L và R rời nhau và tất cả các cạnh trong E nằm giữa L và R. Hình 27.8 minh họa khái niệm của một so khớp.

Bài toán tìm một so khớp cực đại trong một đồ thị hai nhánh có nhiều ứng dụng thức tiễn. Để lấy ví dụ, ta có thể xét việc so khớp một tập hợp



Hình 27.8 Một đồ thị hai nhánh $G = (V, E)$ với phân hoạch đỉnh $V = L \cup R$. **(a)** Một so khớp với bản số 2. **(b)** Một so khớp cực đại với bản số 3.

L máy với một tập hợp R công việc được thực hiện đồng thời.

Ta xem sự hiện diện của cạnh (u, v) trong E có nghĩa là một máy cụ thể $u \in L$ có thể thực hiện một công việc cụ thể $v \in R$. Một so khớp cực đại cung cấp công việc cho càng nhiều máy càng tốt.

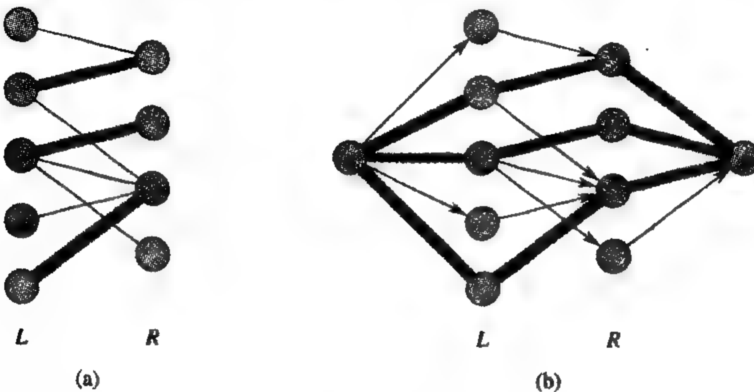
Tìm một so khớp hai nhánh cực đại

Ta có thể dùng phương pháp Ford-Fulkerson để tìm ra một so khớp cực đại trong một đồ thị hai nhánh không hướng $G = (V, E)$ trong thời gian đa thức trong $|V|$ và $|E|$. Điểm tinh tế đó là kiến tạo một mạng luồng ở đó các luồng tương ứng với các lần so khớp, như đã nêu trong Hình 27.9. Ta định nghĩa ***mạng luồng tương ứng*** $G' = (V', E')$ với đồ thị hai nhánh G như sau. Ta cho nguồn s và bồn t là các đỉnh mới không nằm trong V, và ta cho $V' = V \cup \{s, t\}$. Nếu phân hoạch đỉnh của G là $V = L \cup R$, các cạnh có hướng của G' sẽ căn cứ vào

$$E' = \{(s, u) : u \in L\}$$
$$\cup\{(u, v) : u \in L, v \in R, \text{ và } (u, v) \in E\}$$
$$\cup \{(v, t) : v \in R\} .$$

Để hoàn thành phần xây dựng, ta gán dung lượng đơn vị cho mỗi cạnh trong E'.

Định lý dưới đây chứng tỏ một so khớp trong G tương ứng trực tiếp với một luồng trong mạng luồng tương ứng G' của G. Ta nói rằng một luồng f trên một mạng luồng $G = (V, E)$ ***có giá trị số nguyên*** nếu $f(u, v)$ là một số nguyên với tất cả $(u,v) \in V \times V$.



Hình 27.9 Mạng luồng tương ứng với một đồ thị hai nhánh. **(a)** Đồ thị hai nhánh $G = (V, E)$ có phân hoạch đỉnh $V = L \cup R$ từ Hình 27.8. Một so khớp cực đại được nêu bởi các cạnh tô bóng. **(b)** Mạng luồng tương ứng G' có một luồng cực đại đã nêu. Mỗi cạnh có dung lượng đơn vị. Các cạnh tô bóng có một luồng là 1, và tất cả các cạnh khác không mang luồng nào cả. Các cạnh tô bóng từ L đến R tương ứng với các cạnh trong một so khớp cực đại của đồ thị hai nhánh.

Bổ đề 27.10

Cho $G = (V, E)$ là một đồ thị hai nhánh có phân hoạch đỉnh $V = L \cup R$, và cho $G' = (V', E')$ là mạng luồng tương ứng của nó. Nếu M là một so khớp trong G, thì có một luồng có giá trị số nguyên f trong G' với giá trị $|f| = |M|$. Ngược lại, nếu f là một luồng có giá trị số nguyên trong G', thì có một so khớp M trong G với bản số $|M| = |f|$.

Chứng minh Trước tiên ta chứng tỏ a so khớp M trong G tương ứng với một luồng có giá trị số nguyên trong G'. Định nghĩa f như sau. Nếu $(u, v) \in M$, thì $f(s,u) = f(u,v) = f(v,t) = 1$ và $f(u,s) = f(v,u) = f(t,v) = -1$. Với tất cả các cạnh khác $(u, v) \in E'$, ta định nghĩa $f(u, v) = 0$.

Theo trực giác, mỗi cạnh $(u, v) \in M$ tương ứng với 1 đơn vị của luồng trong G' băng ngang lộ trình $s \to u \to v \to t$. Hơn nữa, các lộ trình mà các cạnh trong m cảm sinh có đỉnh rời nhau, ngoại trừ s và t. Để xác minh f thỏa tính đối xứng ghềnh, các hạn chế dung lượng, và sự bảo toàn luồng lưu, ta chỉ cần nhận xét rằng có thể có f bằng cách tăng cường luồng dọc theo mỗi lộ trình như vậy. Luồng mạng qua phần cắt $(L \cup \{s\}, R \cup \{t\})$ bằng với $|M|$; như vậy, theo Bổ đề 27.5, giá trị của luồng là $|f| = |M|$.

Để chứng minh đảo đề, ta cho f là một luồng có giá trị số nguyên trong G' và cho

$$M = \{(u,v) : u \in L,\ v \in R, \text{ và } f(u,v) > 0\}.$$

Mỗi đỉnh $u \in L$ chỉ has một cạnh vào, tức (s, u), và dung lượng của nó là 1. Như vậy, mỗi $u \in L$ có tối đa một đơn vị của luồng mạng dương nhập vào nó. Bởi f có giá trị số nguyên, với mỗi $u \in L$, 1 đơn vị của luồng mạng dương sẽ nhập vào u nếu và chỉ nếu có chính xác một đỉnh $v \in R$ sao cho $f(u, v) = 1$. Như vậy, có tối đa một cạnh rời từng $u \in L$ sẽ mang luồng mạng dương. Có thể tạo một đối số đối xứng cho mỗi $v \in R$. Do đó, tập hợp M được định nghĩa trong phát biểu của định lý là một so khớp.

Để thấy $|M| = |f|$, ta nhận xét rằng với mọi đỉnh so khớp $u \in L$, ta có $f(s, u) = 1$, và với mọi cạnh $(u, v) \in E - M$, ta có $f(u, v) = 0$. Bởi vậy, dùng Bổ đề 27.1, tính đối xứng ghềnh, và không có cạnh nào từ L đến t, ta được

$$\begin{aligned}
|M| &= f(L, R) \\
&= f(L, V') - f(L, L) - f(L,s) - f(L,t) \\
&= 0 - 0 + f(s,L) - 0 \\
&= f(s, V') \\
&= |f|.
\end{aligned}$$

Theo trực giác, một so khớp cực đại trong một đồ thị hai nhánh G tương ứng với một luồng cực đại trong mạng luồng tương ứng G' của nó. Như vậy, ta có thể tính toán một so khớp cực đại trong G bằng cách chạy một thuật toán luồng cực đại trên G'. Khó khăn duy nhất trong biện luận này đó là thuật toán luồng cực đại có thể trả về một luồng trong G' bao gồm các lượng phi tích phân. Định lý dưới đây chứng tỏ nếu ta dùng phương pháp Ford-Fulkerson, khó khăn đó không thể nảy sinh.

Định lý 27.11 (định lý tính tích phân)

Nếu hàm dung lượng c chỉ tiếp nhận các giá trị tích phân, thì luồng cực đại f được tạo bằng phương pháp Ford-Fulkerson sẽ có tính chất: $|f|$ có giá trị số nguyên. Hơn nữa, với tất cả các đỉnh u và v, giá trị của $f(u, v)$ là một số nguyên.

Chứng minh Phần chứng minh sẽ theo phương pháp quy nạp trên số lần lặp lại. Ta để nó làm Bài tập 27.3-2.

Giờ đây, ta có thể chứng minh hệ luận dưới đây theo Bổ đề 27.10.

Hệ luận 27.12

Bản số của một so khớp cực đại trong một đồ thị hai nhánh G là giá trị của một luồng cực đại trong mạng luồng tương ứng G' của nó.

Chứng minh Ta dùng danh pháp trong Bổ đề 27.10. Giả sử rằng M là một so khớp cực đại trong G và luồng tương ứng f trong G' không cực đại. Thì có một luồng cực đại f' trong G' sao cho $|f'| > |f|$. Bởi các dung lượng trong G' có giá trị số nguyên, theo Định lý 27.11, và f cũng vậy. Như vậy, f' tương ứng với một so khớp M' trong G có bản số $|M'| = |f'| > |f| = |M|$, mâu thuẫn với tính cực đại của M. Cũng vậy, ta có thể chứng tỏ nếu f là một luồng cực đại trong G', so khớp tương ứng của nó là một so khớp cực đại trên G.

Như vậy, cho một đồ thị hai nhánh không hướng G, ta có thể tìm ra một so khớp cực đại bằng cách tạo mạng luồng G', chạy phương pháp Ford-Fulkerson, và trực tiếp có được một so khớp cực đại M từ luồng cực đại có giá trị số nguyên f đã tìm thấy. Do bất kỳ so khớp nào trong một đồ thị hai nhánh đều có bản số tối đa $\min(L, R) = O(V)$, nên giá trị của luồng cực đại trong G' là $O(V)$. Do đó, ta có thể tìm một so khớp cực đại trong một đồ thị hai nhánh trong thời gian $O(VE)$.

Bài tập

27.3-1

Chạy thuật toán Ford-Fulkerson trên mạng luồng trong Hình 27.9(b)

và nêu mạng thặng dư sau mỗi phép tăng cường luồng. Đánh số các đỉnh trong L từ trên xuống từ 1 đến 5 và trong R từ trên xuống từ 6 đến 9. Với mỗi lần lặp lại, bạn chọn lộ trình tăng cường nhỏ nhất theo thứ tự từ điển.

27.3-2

Chứng minh Định lý 27.11.

27.3-3

Cho $G = (V, E)$ là một đồ thị hai nhánh với phân hoạch đỉnh $V = L \cup R$, và cho G' là mạng luồng tương ứng của nó. Nêu một cận trên thích hợp trên chiều dài của bất kỳ lộ trình tăng cường nào tìm thấy trong G' trong khi thi hành FORD-FULKERSON.

27.3-4 *

Một ***so khớp hoàn hảo*** [perfect matching] là một so khớp ở đó mọi đỉnh đều ăn khớp. Cho $G = (V, E)$ là một đồ thị hai nhár [illegible] ng hướng có phân hoạch đỉnh $V = L \cup R$, ở đó $|L| = |R|$. Với bất kỳ $X \subseteq V$, hãy định nghĩa ***láng giềng*** của X khi

$N(X) = \{y \in V : (x, y) \in E$ với một $x \in X\}$,

nghĩa là, tập hợp các đỉnh kề với một phần tử của X. Chứng minh ***định lý Hall***: ở đó tồn tại một so khớp hoàn hảo trong G nếu và chỉ nếu $|A| \leq |N(A)|$ với mọi tập hợp con $A \subseteq L$.

27.3-5

Một đồ thị hai nhánh $G = (V, E)$, ở đó $V = L \cup R$, là ***đều-d*** [d-regular] nếu mọi đỉnh $v \in V$ có độ chính xác d. Mọi đồ thị hai nhánh đều - d có $|L| = |R|$. Chứng minh mọi đồ thị hai nhánh đều-d có một so khớp của bản số $|L|$ bằng cách chứng tỏ một phần cắt cực tiểu của mạng luồng tương ứng có dung lượng $|L|$.

---

* 27.4 Các thuật toán đẩy luồng trước

Đoạn này trình bày cách tiếp cận "đẩy luồng trước" [preflow-push] để tính toán các luồng cực đại. Các thuật toán luồng cực đại nhanh nhất cho đến nay là các thuật toán đẩy luồng trước, và các bài toán luồng khác, như bài toán luồng có mức hao phí cực tiểu, có thể được giải quyết một cách hiệu quả bằng các phương pháp đẩy luồng trước. Đoạn này giới thiệu thuật toán luồng cực đại "chung" của Goldberg, có một kiểu thực thi đơn giản chạy trong $O(V^2E)$ thời gian, và do đó cải thiện đối với cận $O(VE^2)$ của thuật toán Edmonds-Karp. Đoạn 27.5 tu chính

thuật toán chung để có được một thuật toán đẩy luồng trước khác chạy trong $O(V^3)$ thời gian.

Các thuật toán đẩy luồng trước làm việc theo cách cục bộ hóa hơn so với phương pháp Ford-Fulkerson. Thay vì xét nguyên cả mạng thặng dư $G = (V, E)$ để tìm ra một lộ trình tăng cường, các thuật toán đẩy luồng trước lần lượt làm việc trên từng đỉnh một, chỉ xem xét các láng giềng của đỉnh trong mạng thặng dư. Vả lại, khác với phương pháp Ford-Fulkerson, các thuật toán đẩy luồng trước không duy trì tính chất bảo toàn luồng lưu xuyên suốt tiến trình thi hành. Tuy nhiên, chúng duy trì một luồng trước [preflow], là một hàm $f : V \times V \rightarrow \mathbf{R}$ thỏa tính đối xứng ghềnh, các hạn chế dung lượng, và phép nới lỏng dưới đây về sự bảo toàn luồng lưu: $f(V, u) \geq 0$ với tất cả các đỉnh $u \in V - \{s\}$. Nghĩa là, ngoài nguồn ra, luồng mạng vào mỗi đỉnh đều không âm. Ta gọi luồng mạng vào một đỉnh u là ***luồng thặng dư*** vào u, căn cứ vào

$$e(u) = f(V, u) . \qquad (27.8)$$

Ta nói rằng một đỉnh $u \in V - \{s, t\}$ đang ***tràn*** nếu $e(u) > 0$.

Để bắt đầu đoạn này, ta mô tả trực giác nằm sau phương pháp đẩy luồng trước. Sau đó, ta sẽ nghiên cứu hai phép toán mà phương pháp sử dụng: "đẩy" luồng trước và "nâng" một đỉnh. Cuối cùng, ta sẽ trình bày một thuật toán đẩy luồng trước chung và phân tích tính đúng đắn và thời gian thực hiện của nó.

Trực giác

Trực giác đàng sau phương pháp đẩy luồng trước được hiểu rõ nhất dưới dạng các luồng chất lỏng: ta xét một mạng luồng $G = (V, E)$ là một hệ thống các ống dẫn tương kết của các dung lượng đã cho. Áp dụng tính tương tự này cho phương pháp Ford-Fulkerson, ta có thể nói rằng mỗi lộ trình tăng cường trong mạng sẽ làm cho một luồng chất lỏng bổ sung nâng lên, mà không có các điểm nhánh, chảy từ nguồn đến bồn. Phương pháp Ford-Fulkerson liên tục bổ sung thêm các luồng chảy cho đến khi không thể bổ sung thêm được nữa.

Thuật toán đẩy luồng trước chung có một trực giác hơi khác. Cũng như trước, các cạnh có hướng tương ứng với các ống dẫn. Các đỉnh, là các khớp nối ống, có hai tính chất đáng quan tâm. Thứ nhất, để điều tiết luồng thặng dư, mỗi đỉnh có một ống thoát dẫn đến một bồn chứa lớn một cách tùy ý có thể tích lũy chất lỏng. Thứ hai, mỗi đỉnh, bồn chứa của nó, và tất cả các tuyến nối ống của nó nằm trên một nền tảng có chiều cao gia tăng khi thuật toán tiến triển.

Các chiều cao đỉnh sẽ xác định luồng sẽ được đẩy ra sao: ta chỉ đẩy luồng đổ xuống đồi, nghĩa là, từ một đỉnh cao hơn xuống một đỉnh thấp

hơn. Có thể có luồng mạng dương từ một đỉnh thấp hơn đến một đỉnh cao hơn, nhưng các phép toán đẩy luồng luôn đẩy nó xuống đồi. Chiều cao của nguồn được cố định tại $|V|$, và chiều cao của bồn được cố định tại 0. Tất cả các chiều cao đỉnh khác bắt đầu tại 0 và gia tăng theo thời gian. Trước tiên, thuật toán gửi càng nhiều luồng càng tốt xuống đồi từ nguồn về phía bồn. Lượng mà nó gửi chính xác đủ để điền mỗi ống đi ra từ nguồn đến dung lượng; nghĩa là, nó gửi dung lượng của phần cắt $(s, V - s)$. Khi lần đầu tiên luồng nhập một đỉnh trung gian, nó gom lại trong bồn chứa của đỉnh. Từ đó, cuối cùng nó được đẩy xuống đồi.

Chung cuộc có thể xảy ra trường hợp các ống dẫn duy nhất rời một đỉnh u và chưa được bão hòa với luồng sẽ nối với các đỉnh nằm trên cùng cấp với u hoặc phía trên thượng nguồn từ u. Trong trường hợp này, để giải thoát một đỉnh tràn u khỏi luồng thặng dư củanó, ta phải gia tăng chiều cao của nó—một phép toán có tên "nâng" đỉnh u. Chiều cao của nó được gia tăng lên thêm một đơn vị so với chiều cao của láng giềng trong số các láng giềng thấp nhất mà nó có một ống dẫn không bão hòa. Do đó, sau khi nâng một đỉnh, ta có ít nhất một ống dẫn ra qua đó có thể đẩy thêm luồng.

Chung cuộc, toàn bộ luồng có thể đi qua đến bồn đều đến đó. Không còn gì có thể đến, bởi các ống dẫn tuân thủ các hạn chế dung lượng; lượng luồng qua bất kỳ phần cắt nào vẫn được hạn chế bởi dung lượng của phần cắt. Để luồng trước [preflow] trở thành một luồng "hợp pháp", thuật toán gửi một phần thặng dư gom lại trong các bồn chứa của các đỉnh tràn trở về nguồn bằng cách tiếp tục nâng các đỉnh lên bên trên chiều cao cố định $|V|$ của nguồn. (Việc vận chuyển phần thặng dư trở về nguồn thực tế được hoàn thành bằng cách khử các luồng gây ra phần thặng dư.) Như sẽ thấy, một khi tất cả các bồn chứa đã được xả trống, luồng trước không những là luồng "hợp pháp", mà còn là một luồng cực đại.

Các phép toán cơ bản

Từ phần mô tả trên đây, ta thấy thuật toán đẩy luồng trước thực hiện hai phép toán cơ bản: đẩy phần thặng dư của luồng từ một đỉnh đến một trong các láng giềng củanó và nâng một đỉnh. Khả năng áp dụng của các phép toán này tùy thuộc vào chiều cao của các đỉnh, mà giờ đây ta định nghĩa một cách chính xác.

Cho $G = (V, E)$ là một mạng luồng với nguồn s và bồn t, và cho f là một luồng trước trong G. Một hàm $h : V \to \mathbf{N}$ là một ***hàm chiều cao*** nếu $h(s) = |V|$, $h(t) = 0$, và

$h(u) \leq h(v) + 1$

với mọi cạnh thặng dư $(u, V) \in E_f$. Ta lập tức có được bổ đề dưới đây.

Bổ đề 27.13

Cho $G = (V, E)$ là một mạng luồng, cho f là một luồng trước trong G, và cho h là một hàm chiều cao trên V. Với hai đỉnh bất kỳ $u, v \in V$, nếu $h(u) > h(v) + 1$, thì (u, v) không phải là một cạnh trong đồ thị thặng dư.

Có thể áp dụng phép toán cơ bản PUSH(u, v) nếu u là một đỉnh tràn, $c_f(u, v) > 0$, và $h(u) = h(v) + 1$. Mã giả dưới đây cập nhật luồng trước f trong một mạng mặc định $G = (V, E)$. Nó mặc nhận rằng các dung lượng được căn cứ vào một hàm thời gian bất biến c và các dung lượng thặng dư cũng có thể được tính toán trong thời gian bất biến đã cho c và f. Luồng thặng dư được lưu trữ tại một đỉnh u được duy trì dưới dạng $e[u]$, và chiều cao của u được duy trì dưới dạng $h[u]$. Biểu thức $d_f(u, v)$ là một biến tạm lưu trữ khối lượng luồng có thể được đẩy từ u đến v.

PUSH(u, v)

1 ▷ **Áp dụng khi:** u tràn, $c_f[u, v] > 0$, và $h[u] = h[v] + 1$.

2 ▷ **Hành động:** Đẩy $d_f(u, v) = \min(e[u], c_f(u, v))$ đơn vị của luồng từ u đến v.

3 $d_f(u, v) \leftarrow \min(e[u], c_f(u, v))$

4 $f[u,v] \leftarrow f[u,v] + d_f(u,v)$

5 $f[v, u] \leftarrow -f[u,v]$

6 $e[u] \leftarrow e[u] - d_f(u, v)$

7 $e[v] \leftarrow e[v] + d_f(u, v)$

Mã của PUSH hoạt động như sau. Đỉnh u được mặc nhận có một phần thặng dư dương $e[u]$, và dung lượng thặng dư của (u, v) là dương. Như vậy, ta có thể vận chuyển tới $d_f(u, v) = \min(e[u], c_f(u, v))$ đơn vị của luồng từ u đến v mà không làm cho $e[u]$ trở thành âm hoặc dung lượng $c(u, v)$ trở thành vượt mức. Giá trị này được tính toán trong dòng 3. Ta dời luồng từ u đến v bằng cách cập nhật f trong các dòng 4-5 và e trong các dòng 6-7. Như vậy, nếu f là một luồng trước trước khi PUSH được gọi, nó vẫn giữ nguyên là một luồng trước sau đấy.

Nhận thấy không có gì trong mã của PUSH tùy thuộc vào các chiều cao của u và v, vả lại ta ngăn cấm triệu gọi nó trừ phi $h[u] = h[v] + 1$. Như vậy, luồng thặng dư chỉ được đẩy xuống đồi theo một vi phân chiều cao là 1. Theo Bổ đề 27.13, không có các cạnh thặng dư tồn tại giữa hai đỉnh có các chiều cao khác nhau hơn 1, và như vậy sẽ không được gì nếu cho phép đẩy luồng xuống đồi theo một vi phân chiều cao trên 1.

Ta gọi phép toán PUSH(u, v) là một ***phép đẩy*** từ u đến v. Nếu áp dụng một phép toán đẩy cho một cạnh (u, v) rời một đỉnh u, ta cũng nói rằng phép toán đẩy áp dụng cho u. Nó là một ***phép đẩy bão hòa*** [saturating push] nếu cạnh (u, v) trở thành ***bão hòa*** ($c_f(u, v) = 0$ sau đấy); bằng không, nó là một ***phép đẩy không bão hòa*** [nonsaturating push]. Nếu một cạnh bão hòa, nó không xuất hiện trong mạng thặng dư.

Phép toán cơ bản LIFT(u) áp dụng nếu u tràn và nếu $c_f(u, v) > 0$ hàm ý $h[u] \le h[v]$ với tất cả các đỉnh v. Nói cách khác, ta có thể nâng một đỉnh tràn u nếu với mọi đỉnh v mà ta có dung lượng thặng dư từ u đến v, luồng không thể được đẩy từ u đến v bởi v không xuống đồi từ u. (Hãy nhớ lại phần định nghĩa, cả nguồn s lẫn bồn t đều không thể tràn, do đó s và t không thể được nâng.)

LIFT(u)

1 ▷ **Áp dụng khi**: u tràn và với tất cả $v \in V$,

$(u, v) \in E_f$ hàm ý $h[u] \le h[v]$.

2 ▷ **Hành động**: Tăng chiều cao của u.

3 $h[u] \leftarrow 1 + \min \{h[v] : (u, v) \in E_f\}$

Khi gọi phép toán LIFT(u), ta nói rằng đỉnh u ***được nâng***. Điều quan trọng cần lưu ý đó là khi u được nâng, E_f phải chứa ít nhất một cạnh rời u, sao cho phép giảm thiểu trong mã nằm trên một tập hợp không trống. Sự việc này là do giả thiết cho rằng u tràn. Bởi $e[u] > 0$, ta có $e[u] := f[V, u] > 0$, và do đó phải có ít nhất một đỉnh v sao cho $f[v, u] > 0$. Vậy thì,

$$\begin{aligned} c_f(u,v) &= c(u,v) - f[u,v] \\ &= c(u,v) + f[v,u] \\ &> 0, \end{aligned}$$

hàm ý rằng $(u, v) \in E_f$. Như vậy, phép toán LIFT(u) cho u chiều cao lớn nhất mà các hạn chế trên các hàm chiều cao cho phép.

Thuật toán chung

Thuật toán đẩy luồng trước chung sử dụng chương trình con dưới đây để tạo một luồng trước ban đầu trong mạng luồng.

```
INITIALIZE-PREFLOW (G, s)
1  for mỗi đỉnh u ∈ V[G]
2       do h[u] ← 0
3          e[u] ← 0
4  for mỗi cạnh (u, v) ∈ E[G]
```

```
            do f[u,v] ← 0
               f[v, u] ← 0
h[s] ← |V[G]|
for mỗi đỉnh u ∈ Adj[s]
     do f[s, u] ← c(s, u)
        f[u, s] ← −c(s, u)
e[u] ← c(s, u)
```

INITIALIZE-PREFLOW tạo một luồng trước ban đầu f được định nghĩa với

$$f[u,v] = \begin{cases} c(u,v) & \text{nếu } u = s, \\ -c(v,u) & \text{nếu } v = s, \\ 0 & \text{bằng không}. \end{cases} \quad (27.9)$$

Nghĩa là, mỗi cạnh rời nguồn được điền vào dung lượng, và tất cả các cạnh khác không mang luồng nào cả. Với mỗi đỉnh v kề với nguồn, thoạt đầu ta có $e[v] = c(s, v)$. Thuật toán chung cũng bắt đầu bằng một hàm chiều cao ban đầu h, căn cứ vào

$$h[u] = \begin{cases} |V| & \text{nếu } u = s, \\ 0 & \text{bằng không}. \end{cases}$$

Đây là một hàm chiều cao bởi các cạnh chỉ (u, v) mà $h[u] > h[v] + 1$ là những cạnh mà $u = s$, và những cạnh được bão hòa, có nghĩa là chúng không nằm trong mạng thặng dư.

Thuật toán dưới đây là ví dụ tiêu biểu cho phương pháp đẩy luồng trước.

```
GENERIC-PREFLOW-PUSH(G)
INITIALIZE-PREFLOW(G, s)
while ở đó tồn tại một phép đẩy hoặc phép toán nâng có thể áp dụng
   do lựa một phép đẩy hoặc phép nâng có thể áp dụng và thực hiện nó
```

Sau khi khởi tạo luồng, thuật toán chung liên tục áp dụng các phép toán cơ bản mọi nơi có thể, theo một thứ tự bất kỳ. Bổ đề dưới đây cho biết miễn là có một đỉnh tràn tồn tại, ít nhất một trong hai phép toán được áp dụng.

***Bổ đề 27.14** (Một đỉnh tràn có thể được đẩy hoặc nâng)*

Cho $G = (V, E)$ là một mạng luồng có nguồn s và bồn t, cho f là một

luồng trước, và cho h là một hàm chiều cao bất kỳ cho f. Nếu u là một đỉnh tràn bất kỳ, thì một phép đẩy hoặc phép nâng áp dụng cho nó.

Chứng minh Với một cạnh thặng dư (u, v), ta có $h(u) \le h(v) + 1$ bởi h là một hàm chiều cao. Nếu một phép đẩy không áp dụng cho u, thì với tất cả các cạnh thặng dư (u, v), ta phải có $h(u) < h(v) + 1$, hàm ý $h(u) \le h(v)$. Như vậy, một phép nâng có thể áp dụng cho u.

Tính đúng đắn của phương pháp đẩy luồng trước

Để chứng tỏ thuật toán đẩy luồng trước chung giải quyết bài toán luồng cực đại, trước tiên ta chứng minh nếu nó kết thúc, luồng trước f là một luồng cực đại. Về sau ta sẽ chứng minh nó kết thúc. Để bắt đầu, ta nêu vài nhận xét về hàm chiều cao h.

Bổ đề 27.15 (Các chiều cao đỉnh không bao giờ giảm)

Trong khi thi hành GENERIC-PREFLOW-PUSH trên một mạng luồng $G = (V, E)$, với mỗi đỉnh $u \in V$, chiều cao $h[u]$ không bao giờ giảm. Hơn nữa, mỗi khi một phép nâng được áp dụng cho một đỉnh u, chiều cao của nó $h[u]$ gia tăng ít nhất là 1.

Chứng minh Bởi các chiều cao đỉnh chỉ thay đổi trong các phép nâng, nên nó đủ để chứng minh phát biểu thứ hai của bổ đề. Nếu đỉnh u được nâng, thì với tất cả các đỉnh v sao cho $(u, v) \in E_f$ ta có $h[u] \le h[v]$; điều này hàm ý $h[u] < 1 + \min\{h[v] : (u, v) \in E_f\}$, và do đó phép toán phải tăng $h[u]$.

Bổ đề 27.16

Cho $G = (V, E)$ là một mạng luồng với nguồn s và bồn t. Trong khi thi hành GENERIC-PREFLOW-PUSH trên G, thuộc tính h được duy trì dưới dạng một hàm chiều cao.

Chứng minh Phần chứng minh theo phương pháp quy nạp trên số lượng các phép toán cơ bản được thực hiện. Thoạt đầu, h là một hàm chiều cao, như đã nhận xét.

Ta biện luận rằng nếu h là một hàm chiều cao, thì một phép toán LIFT(u) để lại cho h một hàm chiều cao. Nếu ta xem xét một cạnh thặng dư $(u, v) \in E_f$ rời u, thì phép toán LIFT(u) bảo đảm $h[u] \le h[v] + 1$ sau đó. Giờ đây, ta xét một cạnh thặng dư (w, u) nhập u. Theo Bổ đề 27.15, $h[w] \le h[u] + 1$ trước phép toán LIFT(u) hàm ý $h[w] < h[u] + 1$ sau đó. Như vậy, phép toán LIFT(u) để lại cho h một hàm chiều cao.

Giờ đây, xét một phép toán PUSH(u, v). Phép toán này có thể bổ sung cạnh (v, u) vào E_f và nó có thể gỡ bỏ (u, v) ra khỏi E_f. Trong trường hợp trước, ta có $h[v] = h[u] - 1$, và do đó h vẫn là một hàm chiều

cao. Trong trường hợp sau, việc gỡ bỏ (u, v) ra khỏi mạng thặng dư sẽ gỡ bỏ sự ràng buộc tương ứng, và h lại vẫn là một hàm chiều cao.

Bổ đề dưới đây cho ta một tính chất quan trọng về các hàm chiều cao.

Bổ đề 27.17

Cho $G = (V, E)$ là một mạng luồng với nguồn s và bồn t, cho f là một luồng trước trong G, và cho h là một hàm chiều cao trên V. Như vậy, không có lộ trình nào từ nguồn s đến bồn t trong mạng thặng dư G_f.

Chứng minh Vì sự mâu thuẫn, ta giả sử có một lộ trình $p = (v_0, v_1, \ldots, v_k)$ từ s đến t trong G_f, ở đó $v_0 = s$ và $v_k = t$. Không để mất tính tổng quát, p là một lộ trình đơn giản, và do đó $k < |V|$. Với $i = 0, 1, \ldots, k - 1$, cạnh $(v_i, v_{i+1}) \in E_f$. Bởi h là một hàm chiều cao, $h(v_i) \leq h(v_{i+1}) + 1$ với $i = 0, 1, \ldots, k - 1$. Tổ hợp các bất đẳng thức này trên lộ trình p cho ra $h(s) \leq h(t) + k$. Nhưng do $h(t) = 0$, nên ta có $h(s) \leq k < |V|$, mâu thuẫn với yêu cầu rằng $h(s) = |V|$ trong một hàm chiều cao.

Giờ đây, ta đã sẵn sàng chứng tỏ nếu thuật toán đẩy luồng trước chung kết thúc, luồng trước mà nó tính toán là một luồng cực đại.

***Định lý* 27.18 (*Tính đúng đắn của thuật toán đẩy luồng trước chung*)**

Nếu thuật toán GENERIC-PREFLOW-PUSH kết thúc khi chạy trên một mạng luồng $G = (V, E)$ với nguồn s và bồn t, thì luồng trước f mà nó tính toán là một luồng cực đại cho G.

Chứng minh Nếu thuật toán chung kết thúc, thì mỗi đỉnh trong $V - \{s, t\}$ phải có một phần thặng dư 0, bởi theo các Bổ đề 27.14 và 27.16 và sự bất biến f luôn là một luồng trước, nên không có các đỉnh tràn. Do đó, f là một luồng. Bởi h là một hàm chiều cao, nên theo Bổ đề 27.17 sẽ không có lộ trình nào từ s đến t trong mạng thặng dư G_f. Do đó, theo định lý max-flow min-cut, f là một luồng cực đại.

Phân tích phương pháp đẩy luồng trước

Để chứng tỏ thuật toán đẩy luồng trước chung quả thực kết thúc, ta sẽ định cận số lượng phép toán mà nó thực hiện. Mỗi trong ba kiểu phép toán—nâng, đẩy bão hòa, và đẩy không bão hòa—được định cận tách biệt. Với kiến thức về các cận này, nó là một bài toán đơn giản để kiến tạo một thuật toán chạy trong $O(V^2E)$ thời gian. Tuy nhiên, trước khi bắt đầu phân tích, ta chứng minh một bổ đề quan trọng.

Bổ đề 27.19

Cho $G = (V, E)$ là một mạng luồng với nguồn s và bồn t, và cho f là

một luồng trước trong G. Như vậy, với một đỉnh tràn u, ta có một lộ trình đơn giản từ u đến s trong mạng thặng dư G_f.

Chứng minh Cho $U = \{v$: ở đó tồn tại một lộ trình đơn giản từ u đến v trong $G_f\}$, và vì sự mâu thuẫn giả sử rằng $s \notin U$. Cho $\overline{U} = V - U$.

Ta biện luận với mỗi cặp đỉnh $v \in U$ và $w \in \overline{U}$ rằng $f(w,v) \leq 0$. Tại sao? Nếu $f(w, v) > 0$, thì $f(v, w) < 0$, hàm ý rằng $c_f(v, w) = c(v, w) - f(v, w) > 0$. Như vậy, ở đó tồn tại một cạnh $(v, w) \in E_f$, và do đó một lộ trình đơn giản theo dạng $u \rightsquigarrow v \rightarrow w$ trong G_f, mâu thuẫn với chọn lựa của chúng ta về w.

Như vậy, ta phải có $f(\overline{U}, U) \leq 0$, bởi mọi số hạng trong phép lấy tổng ẩn này không âm. Như vậy, từ phương trình (27.8) và Bổ đề 27.1, ta có thể kết luận

$$\begin{aligned} e(U) &= f(V, U) \\ &= f(\overline{U}, U) + f(U, U) \\ &= f(\overline{U}, U) \\ &\leq 0. \end{aligned}$$

Các phần dư không âm với tất cả các đỉnh trong $V - \{s\}$; bởi ta đã mặc nhận $U \subseteq V - \{s\}$, do đó ta phải có $e(v) = 0$ với tất cả các đỉnh $v \in U$. Nói cụ thể, $e(u) = 0$, mâu thuẫn với giả thiết cho rằng u tràn.

Bổ đề kế tiếp định cận các chiều cao của các đỉnh, và hệ luận của nó định cận tổng số lượng phép nâng được thực hiện.

Bổ đề 27.20

Cho $G = (V, E)$ là một mạng luồng với nguồn s và bồn t. Tại bất kỳ thời điểm nào trong khi thi hành GENERIC-PREFLOW-PUSH trên G, ta có $h[u] \leq 2|V| - 1$ với tất cả các đỉnh $u \in V$.

Chứng minh Các chiều cao của nguồn s và bồn t không bao giờ thay đổi bởi theo định nghĩa các đỉnh này không tràn. Như vậy, ta luôn có $h[s] = |V|$ và $h[t] = 0$.

Bởi một đỉnh chỉ được nâng khi nó tràn, nên ta có thể xét một đỉnh tràn $u \in V - \{s, t\}$. Bổ đề 27.19 cho biết có một lộ trình đơn giản p từ u đến s trong G_f. Cho $p = \langle v_0, v_1, \ldots, v_k \rangle$, ở đó $v_0 = u$, $v_k = s$, và $k \leq |V| - 1$ bởi p là đơn giản. Với $i = 0, 1, \ldots, k - 1$, ta có $(v_i, v_{i+1}) \in E_f$, và do đó, theo Bổ đề 27.16, $h[v_i] \leq h[v_{i+1}] + 1$.

Mở rộng các bất đẳng thức này trên lộ trình p cho ra $h[u] = h[v_0] \leq h[v_k] + k \leq h[s] + (|V| - 1) = 2|V| - 1$.

***Hệ luận 27.21** (**Định cận trên các phép nâng**)*

Cho $G = (V, E)$ là một mạng luồng với nguồn s và bồn t. Như vậy, trong khi thi hành GENERIC-PREFLOW-PUSH trên G, số lượng các phép nâng tối đa là $2|V| - 1$ mỗi đỉnh và tối đa là $(2|V| - 1)(|V| - 2) < 2|V|^2$ nói chung.

Chứng minh Chỉ các đỉnh trong $V - \{s, t\}$, có số $|V| - 2$, có thể được nâng. Cho $u \in V - \{s, t\}$. Phép toán LIFT(u) gia tăng $h[u]$. Giá trị của $h[u]$ thoạt đầu là 0 và theo Bổ đề 27.20 tăng trưởng tối đa là $2|V| - 1$. Như vậy, mỗi đỉnh $u \in V - \{s, t\}$ được nâng tối đa $2|V| - 1$ lần, và tổng số phép nâng được thực hiện tối đa là $(2|V| - 1)(|V| - 2) < 2|V|^2$.

Bổ đề 27.20 cũng giúp ta định cận số lượng phép đẩy bão hòa.

***Bổ đề 27.22** (**Định cận trên các đẩy bão hòa**)*

Trong khi thi hành GENERIC-PREFLOW-PUSH trên một mạng luồng $G = (V, E)$, số lượng các phép đẩy bão hòa tối đa là $2|V||E|$.

Chứng minh Với một cặp đỉnh $u, v \in V$, xét các phép đẩy bão hòa từ u đến v và từ v đến u. Nếu có một phép đẩy như vậy, ít nhất một trong số (u, v) và (v, u) thực tế là một cạnh trong E. Giờ đây, ta giả sử xảy ra một phép đẩy bão hòa từ u đến v. Để một phép đẩy khác từ u đến v xảy ra về sau, trước tiên thuật toán phải đẩy luồng từ v đến u, là điều không thể xảy ra cho đến khi $h[v]$ gia tăng ít nhất là 2. Cũng vậy, $h[u]$ phải gia tăng ít nhất là 2 giữa các phép đẩy bão hòa từ v đến u.

Xét dãy A các số nguyên căn cứ vào $h[u] + h[v]$ với mỗi phép đẩy bão hòa xảy ra giữa các đỉnh u và v. Ta muốn định cận chiều dài của dãy này. Khi xảy ra phép đẩy đầu tiên theo một trong hai hướng giữa u *và* v, ta phải có $h[u] + h[v] \geq 1$; như vậy, số nguyên đầu tiên trong A ít nhất là 1. Khi xảy ra phép đẩy cuối như vậy, ta có $h[u] + h[v] \leq (2|V| - 1) + (2|V| - 2) = 4|V| - 3$ theo Bổ đề 27.20; như vậy, số nguyên cuối trong A tối đa là $4|V| - 3$. Theo đối số trong đoạn trên đây, tối đa mọi số nguyên khác có thể xảy ra trong A. Như vậy, số lượng các số nguyên trong A tối đa là $((4|V| - 3) - 1)/2 + 1 = 2|V| - 1$. (Ta cộng 1 để bảo đảm cả hai đầu của dãy đều được đếm.) Do đó, tổng các phép đẩy bão hòa giữa các đỉnh u *và* v tối đa là $2|V| - 1$. Nhân với số lượng các cạnh sẽ cho ra một tổng số các phép đẩy bão hòa tối đa là $(2|V| - 1)|E| < 2|V||E|$.

Bổ đề dưới đây định cận số lượng các phép đẩy không bão hòa trong thuật toán đẩy luồng trước chung.

Bổ đề 27.23 (Định cận trên các phép đẩy không bão hòa)

Trong khi thi hành GENERIC-PREFLOW-PUSH trên một mạng luồng

$G = (V, E)$, số lượng phép đẩy không bão hòa tối đa là $4|V|^2(|V| + |E|)$.

Chứng minh Định nghĩa một hàm thế $\Phi = \Sigma_{v \in X} h[v]$, ở đó $X \subseteq V$ là tập hợp các đỉnh tràn. Thoạt đầu, $\Phi = 0$. Nhận thấy việc nâng một đỉnh u làm tăng Φ tối đa là $2|V|$, bởi tập hợp để lấy tổng là giống nhau và u không thể được nâng nhiều hơn chiều cao khả dĩ cực đại của nó, mà, theo Bổ đề 27.20, tối đa là $2|V|$. Ngoài ra, một phép đẩy bão hòa từ một đỉnh u đến một đỉnh v làm tăng Φ tối đa là $2|V|$, bởi không có chiều cao nào thay đổi và chỉ có đỉnh v, mà chiều cao của nó tối đa là $2|V|$, mới có khả năng trở thành tràn. Cuối cùng, nhận thấy một phép đẩy không bão hòa từ u đến v giảm Φ ít nhất là 1, bởi u không còn tràn sau phép đẩy, v tràn sau đó cho dù nó không sẵn sàng trước, và $h[v] - h[u] = -1$.

Như vậy, trong khi thi hành thuật toán, tổng lượng tăng trong Φ bị ràng buộc bởi Hệ luận 27.21 và Bổ đề 27.22 tối đa là $(2|V|)(2|V|^2) + (2|V|)(2|V||E|) = 4|V|^2(|V| + |E|)$. Do $\Phi \geq 0$, tổng lượng giảm, và do đó tổng các phép đẩy không bão hòa, tối đa là $4|V|^2(|V| + |E|)$.

Giờ đây, ta đã ấn định giai đoạn cho phần phân tích dưới đây của thủ tục GENERIC-PREFLOW-PUSH, và do đó của bất kỳ thuật toán nào dựa trên phương pháp đẩy luồng trước.

Định lý 27.24

Trong khi thi hành GENERIC-PREFLOW-PUSH trên một mạng luồng $G = (V, E)$, số lượng phép toán cơ bản là $O(V^2E)$.

Chứng minh Tức thời từ Hệ luận 27.21 và các Bổ đề 27.22 và 27.23.

Hệ luận 27.25

Có một thực thi của thuật toán đẩy luồng trước chung chạy trong $O(V^2E)$ thời gian trên một mạng luồng $G = (V, E)$.

Chứng minh Bài tập 27.4-1 yêu cầu bạn nêu cách thực thi thuật toán chung với một phần việc chung là $O(V)$ trên mỗi phép nâng và $O(1)$ trên mỗi phép đẩy. Như vậy, hệ luận đứng vững.

Bài tập

27.4-1

Nêu cách thực thi thuật toán đẩy luồng trước chung dùng $O(V)$ thời gian trên mỗi phép nâng và $O(1)$ thời gian trên mỗi phép đẩy với một tổng thời gian $O(V^2E)$.

27.4-2

Chứng minh thuật toán đẩy luồng trước chung trải qua chỉ tổng cộng

$O(VE)$ thời gian để thực hiện tất cả $O(V^2)$ phép nâng.

27.4-3

Giả sử đã tìm thấy một luồng cực đại trong một mạng luồng $G = (V, E)$ dùng một thuật toán đẩy luồng trước. Nêu một thuật toán nhanh để tìm phần cắt cực tiểu trong G.

27.4-4

Nêu một thuật toán đẩy luồng trước hiệu quả để tìm một so khớp cực đại trong một đồ thị hai nhánh. Phân tích thuật toán của bạn.

27.4-5

Giả sử rằng tất cả các dung lượng cạnh trong một mạng luồng $G = (V, E)$ nằm trong tập hợp $\{1, 2, ..., k\}$. Phân tích thời gian thực niện của thuật toán đẩy luồng trước chung theo dạng $|V|$, $|E|$, và k. (*Mách nước*: Mỗi cạnh có thể hỗ trợ một phép đẩy không bão hòa bao nhiêu lần trước khi nó trở thành bão hòa?)

27.4-6

Chứng tỏ dòng 7 của INITIALIZE-PREFLOW có thể thay đổi thành

$h[s] \leftarrow |V[G]| - 2$

mà không ảnh hưởng đến tính đúng đắn hoặc khả năng thực hiện tiệm cận của thuật toán đẩy luồng trước chung.

27.4-7

Cho $\delta_f(u, v)$ là khoảng cách (số cạnh) từ u đến v trong mạng thặng dư G_f. Chứng tỏ GENERIC-PREFLOW-PUSH duy trì các tính chất cho rằng $h[u] < |V|$ hàm ý $h[u] \leq \delta_f(u, t)$ và $h[u] \geq |V|$ hàm ý $h[u] - |V| \leq \delta_f(u, s)$.

27.4-8 *

Như trong bài tập trên đây, cho $\delta_f(u,v)$ là khoảng cách từ u đến v trong mạng thặng dư G_f. Nêu cách sửa đổi thuật toán đẩy luồng trước chung để duy trì tính chất cho rằng $h[u] < |V|$ hàm ý $h[u] = \delta_f(u, t)$ và $h[u] \geq |V|$ hàm ý $h[u] - |V| = \delta_f(u, s)$. Tổng thời gian mà kiểu thực thi của bạn công hiến để duy trì tính chất này phải là $O(VE)$.

27.4-9

Chứng tỏ số lượng các phép đẩy không bão hòa được GENERIC-PREFLOW-PUSH thi hành trên một mạng luồng $G = (V, E)$ tối đa là $4|V|^2|E|$ với $|V| \geq 4$.

* 27.5 Thuật toán nâng tới trước

Phương pháp đẩy luồng trước cho phép ta áp dụng các phép toán cơ bản theo một thứ tự bất kỳ. Tuy nhiên, nếu chọn thứ tự cẩn thận và quản lý mạng cấu trúc dữ liệu một cách hiệu quả, ta có thể giải quyết bài toán luồng cực đại nhanh hơn cận $O(V^2E)$ căn cứ vào Hệ luận 27.25. Giờ đây, ta sẽ xét thuật toán nâng tới trước, một thuật toán đẩy luồng trước có thời gian thực hiện là $O(V^3)$, mà theo tiệm cận ít nhất nó cũng tốt bằng $O(V^2E)$.

Thuật toán nâng tới trước duy trì một danh sách của các đỉnh trong mạng. Bắt đầu tại trước [front], thuật toán quét danh sách, liên tục lựa một đỉnh tràn u rồi "xả" nó, nghĩa là, thực hiện các phép đẩy và nâng cho đến khi u không còn có một phần thặng dư dương. Mỗi khi một đỉnh được nâng, nó được dời lên đầu danh sách (do đó có tên "nâng tới trước") và thuật toán bắt đầu lại lần quét của nó.

Tính đúng đắn và khả năng phân tích của thuật toán nâng tới trước sẽ tùy thuộc vào khái niệm của các cạnh "chấp nhận được": các cạnh trong mạng thặng dư qua đó luồng có thể được đẩy. Sau khi chứng minh vài tính chất về mạng của các cạnh chấp nhận được, ta sẽ nghiên cứu phép toán xả rồi trình bày và phân tích chính thuật toán nâng tới trước.

Các cạnh và các mạng chấp nhận được

Nếu $G = (V, E)$ là một mạng luồng có nguồn s và bồn t, f là một luồng trước trong G, và h là một hàm chiều cao, thì ta nói rằng (u, v) là một ***cạnh chấp nhận được*** nếu $c_f(u, v) > 0$ và $h(u) = h(v) + 1$. Bằng không, (u, v) là ***không chấp nhận được***. ***Mạng chấp nhận được*** là $G_{f,h} = (V, E_{f,h})$, ở đó $E_{f,h}$ là tập hợp các cạnh chấp nhận được.

Mạng chấp nhận được bao gồm các cạnh qua đó luồng có thể được đẩy. Bổ đề dưới đây chứng tỏ mạng này là một đồ thị phi chu trình có hướng (dag).

Bổ đề 27.26 (Mạng chấp nhận được là phi chu trình)

Nếu $G = (V, E)$ là một mạng luồng, f là một luồng trước trong G, và h là một hàm chiều cao trên G, thì mạng chấp nhận được $G_{f,h} = (V, E_{f,h})$ là phi chu trình.

Chứng minh Phần chứng minh theo sự mâu thuẫn. Giả sử $G_{f,h}$ chứa một chu trình $p = (v_0, v_1, ..., v_k)$, ở đó $v_0 = v_k$ và $k > 0$. Bởi mỗi cạnh trong p là chấp nhận được, ta có $h(v_{i-1}) = h(v_i) + 1$ với $i = 1, 2, ..., k$. Tổng cộng quanh chu trình cho ta

$$\sum_{i=1}^{k} h(v_{i-1}) = \sum_{i=1}^{k} (h(v_i) + 1)$$
$$= \sum_{i=1}^{k} h(v_i) + k .$$

Bởi mỗi đỉnh trong chu trình p xuất hiện một lần trong mỗi phép lấy tổng, ta suy ra sự mâu thuẫn rằng $0 = k$.

Hai bổ đề kế tiếp nêu cách thức mà các phép đẩy và nâng thay đổi mạng chấp nhận được.

Bổ đề 27.27

Cho $G = (V, E)$ là một mạng luồng, cho f là một luồng trước trong G, và cho h là một hàm chiều cao. Nếu một đỉnh u tràn và (u, v) là một cạnh chấp nhận được, thì PUSH(u, v) được áp dụng. Phép toán không tạo bất kỳ cạnh mới nào chấp nhận được, nhưng nó có thể khiến (u, v) trở thành không chấp nhận được.

Chứng minh Theo phần định nghĩa của một cạnh chấp nhận được, luồng có thể được đẩy từ u đến v. Bởi u tràn, phép toán PUSH(u, v) được áp dụng. Cạnh thặng dư mới duy nhất có thể được tạo bằng cách đẩy luồng từ u đến v là cạnh (v, u). Bởi $h(v) = h(u) - 1$, nên cạnh (v, u) không thể trở thành chấp nhận được. Nếu phép toán là một phép đẩy bão hòa, thì $c_f(u, v) = 0$ sau đó và (u, v) trở thành không chấp nhận được.

Bổ đề 27.28

Cho $G = (V, E)$ là một mạng luồng, cho f là một luồng trước trong G, và cho h là một hàm chiều cao. Nếu một đỉnh u tràn và không có cạnh chấp nhận được rời u, thì LIFT(u) được áp dụng. Sau phép nâng, ta có ít nhất một cạnh chấp nhận được rời u, nhưng không có các cạnh chấp nhận được nhập u.

Chứng minh Nếu u tràn, thì theo Bổ đề 27.14, hoặc một phép đẩy hoặc một phép nâng được áp dụng cho nó. Nếu không có các cạnh chấp nhận được rời u, không có luồng nào được đẩy từ u và LIFT(u) được áp dụng. Sau phép nâng, $h[u] = 1 + \min\{h[v] : (u, v) \in E_f\}$. Như vậy, nếu v là một đỉnh thực hiện cực tiểu trong tập hợp này, cạnh (u, v) trở thành chấp nhận được. Do đó, sau phép nâng, có ít nhất một cạnh chấp nhận được rời u.

Để chứng tỏ không có cạnh chấp nhận được nhập u sau một phép nâng, ta giả sử có một đỉnh v sao cho (v, u) chấp nhận được. Như vậy, $h[v] = h[u] + 1$ sau phép nâng, và do đó $h[v] > h[u] + 1$ ngay trước phép nâng. Nhưng theo Bổ đề 27.13, không có các cạnh thặng dư tồn tại giữa các đỉnh mà các chiều cao của chúng khác nhau hơn 1. Hơn nữa, việc

nâng một đỉnh sẽ không làm thay đổi mạng thặng dư. Như vậy, (v, u) không nằm trong mạng thặng dư, và do đó nó không thể nằm trong mạng chấp nhận được.

Các danh sách láng giềng

Các cạnh trong thuật toán nâng tới trước được tổ chức thành "các danh sách láng giềng." Cho một mạng luồng $G = (V, E)$, ***danh sách láng giềng*** $N[u]$ với một đỉnh $u \in V$ là một danh sách nối kết đơn gồm các láng giềng của u trong G. Như vậy, đỉnh v xuất hiện trong danh sách $N[u]$ nếu $(u, v) \in E$ hoặc $(v, u) \in E$. Danh sách láng giềng $N[u]$ chứa chính xác các đỉnh v mà ta có thể có một cạnh thặng dư (u, v). Đỉnh đầu tiên trong $N[u]$ được trỏ đến bởi *head*$[N[u]]$. Đỉnh theo sau v trong một danh sách láng giềng được trỏ đến bởi *next-neighbor*$[v]$; biến trỏ này là NIL nếu v là đỉnh cuối trong danh sách láng giềng.

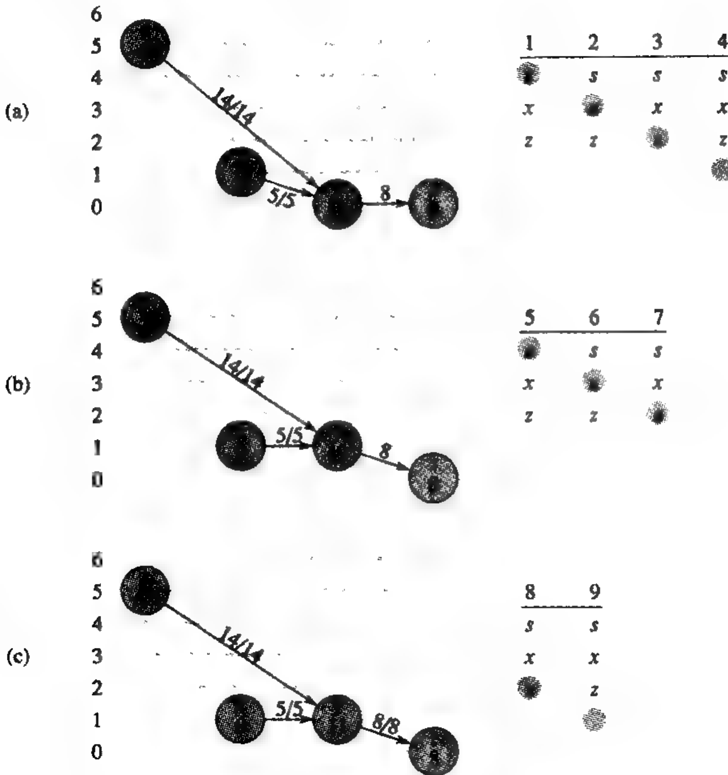
Thuật toán nâng tới trước duyệt vòng qua từng danh sách láng giềng theo thứ tự tùy ý được cố định xuyên suốt tiến trình thi hành thuật toán. Với mỗi đỉnh u, trường *current*$[u]$ trỏ đến đỉnh hiện đang được xem xét trong $N[u]$. Thoạt đầu, *current*$[u]$ được ấn định là *head*$[N[u]]$.

Xả một đỉnh tràn

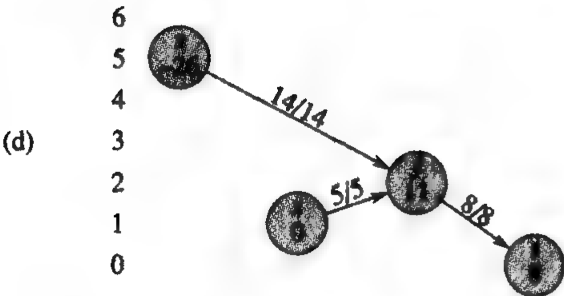
Một đỉnh tràn u được ***xả*** [discharged] bằng cách đẩy tất cả toàn bộ luồng thặng dư của nó qua các cạnh chấp nhận được đến các đỉnh lân cận, nâng u khi cần để khiến các cạnh rời u trở thành chấp nhận được. Mã giả có dạng như sau.

```
DISCHARGE(u)
1 while e[u] > 0
2     do v ← current[u]
3        if v = NIL
4           then LIFT(u)
5                current[u] ← head[N[u]]
6        else if c_f(u, v) > 0 và h[u] = h[v] + 1
7           then PUSH(u, v)
8        else current[u] ← next-neighbor[v]
```

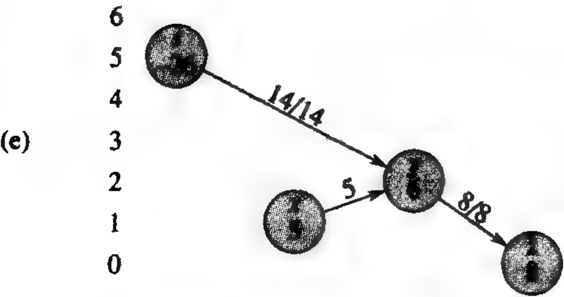
Hình 27.10 rà qua vài lần lặp lại của vòng lặp **while** trong các dòng 1-8, thi hành bao lâu đỉnh u có phần thặng dư dương. Mỗi lần lặp lại sẽ thực hiện chính xác một trong ba hành động, tùy thuộc vào đỉnh v hiện hành trong danh sách láng giềng $N[u]$.



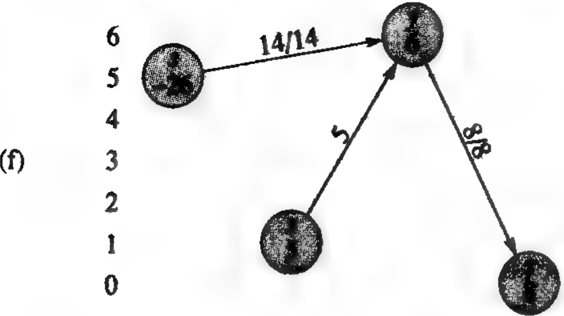
Hình 27.10 Xả một đỉnh. Nó trải qua 15 lần lặp lại của vòng lặp **while** của DISCHARGE để đẩy toàn bộ luồng thặng dư từ đỉnh *y*. Chỉ những láng giềng của *y* và các cạnh nhập hoặc rời *y* mới được nêu. Trong mỗi phần, con số bên trong mỗi đỉnh là phần thặng dư củanó tại đầu của lần lặp lại đầu tiên nêu trong phần đó, và mỗi đỉnh được nêu theo chiều cao củanó xuyên suốt phần. Về bên phải danh sách láng giềng *N*[*y*] được nêu tại đầu mỗi lần lặp lại, với số lần lặp lại nằm trên đầu. Láng giềng được tô bóng là *current*[*y*]. **(a)** Thoạt đầu, có 19 đơn vị của phần thặng dư để đẩy từ *y*, và *current*[*y*] = *s*. Các lần lặp lại 1, 2, và 3 chỉ việc đưa *current*[*y*] ra phía trước, bởi không có các cạnh chấp nhận được rời *y*. Trong lần lặp lại 4, *current*[*y*] = NIL (được nêu bằng kiểu tô bóng nằm bên dưới danh sách láng giềng), và do đó *y* được nâng và *current*[*y*] được chỉnh lại về đầu của danh sách láng giềng. **(b)** Sau phép nâng, đỉnh *y* có chiều cao 1. Trong các lần lặp lại 5 và 6, các cạnh (*y*, *s*) và (*y*, *x*) được tìm thấy là không chấp nhận được, nhưng 8 đơn vị của luồng thặng dư được đẩy từ *y* đến *z* trong lần lặp lại 7. Bởi phép đẩy, nên *current*[*y*] không được đưa ra phía trước trong lần lặp lại này. **(c)** Bởi phép đẩy trong lần lặp lại 7 đã bão hòa cạnh (*y*, *z*), nên nó được tìm thấy là không chấp nhận được trong lần lặp lại 8. Trong lần lặp lại 9, *current*[*y*] = NIL, và do đó đỉnh *y* lại được nâng và *current*[*y*] được chỉnh lại. **(d)** Trong lần lặp lại 10, (*y*, *s*) không chấp nhận được, nhưng 5 đơn vị của luồng thặng dư được đẩy từ *y* đến *x* trong lần lặp lại 11. **(e)** Bởi *current*[*y*] không được đưa ra phía trước trong lần lặp lại 11, nên lần lặp lại 12 thấy (*y*, *x*) là không chấp nhận được. Lần lặp lại 13 thấy (*y*, *z*) là không chấp nhận được, và lần lặp lại 14 nâng đỉnh *y* và chỉnh lại *current*[*y*]. **(f)** Lần lặp lại 15 đẩy 6 đơn vị của luồng thặng dư từ *y* đến *s*. **(g)** Đỉnh *y* giờ đây không có luồng thặng dư, và DISCHARGE kết thúc. Trong ví dụ này, DISCHARGE bắt đầu và hoàn tất với biến trỏ hiện hành tại đầu của danh sách láng giềng, nhưng nói chung điều này không nhất thiết phải như vậy.



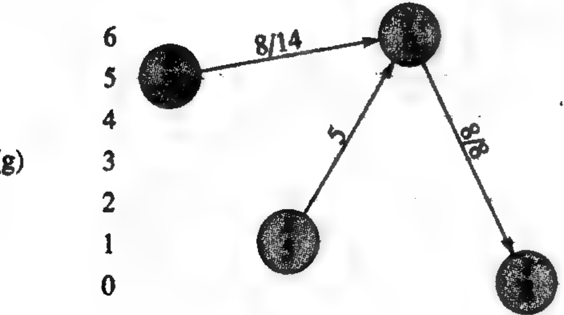
10	11
[illegible]	s
x	[illegible]
z	z



12	13	14
s	s	s
[illegible]	x	x
z	[illegible]	z
		[illegible]



15
[illegible]
x
z



1. Nếu v là NIL, ta đã chạy ra ngoài cuối $N[u]$. Dòng 4 nâng đỉnh u, rồi dòng 5 chỉnh lại láng giềng hiện hành của u trở thành đầu tiên trong $N[u]$. (Bổ đề 27.29 dưới đây phát biểu phép nâng áp dụng trong tình huống này.)

2. Nếu v không NIL và (u, v) là một cạnh chấp nhận được (được xác định bởi đợt kiểm tra trong dòng 6), thì dòng 7 đẩy một phần (hoặc có thể tất cả) phần thặng dư của u đến đỉnh v.

3. Nếu v không NIL nhưng (u, v) là không chấp nhận được, thì dòng 8 đưa $current[u]$ ra phía trước một vị trí trong danh sách láng giềng $N[u]$.

Nhận thấy nếu DISCHARGE được gọi trên một đỉnh tràn u, thì hành động cuối được thực hiện bởi DISCHARGE phải là một phép đẩy từ u. Tại sao? Thủ tục chỉ kết thúc khi $e[u]$ trở thành zero, và phép nâng hoặc sự tiến tới của biến trỏ $current[u]$ không tác động đến giá trị của $e[u]$.

Ta phải bảo đảm khi PUSH hoặc LIFT được DISCHARGE gọi, phép toán được áp dụng. Bổ đề dưới đây chứng minh sự việc này.

Bổ đề 27.29

Nếu DISCHARGE gọi PUSH(u,v) trong dòng 7, thì một phép đẩy áp dụng cho (u, v). Nếu DISCHARGE gọi LIFT(u) trong dòng 4, thì một phép nâng áp dụng cho u.

Chứng minh Các đợt kiểm tra trong các dòng 1 và 6 bảo đảm một phép đẩy chỉ xảy ra nếu phép toán áp dụng, chứng minh phát biểu đầu tiên trong bổ đề.

Để chứng minh phát biểu thứ hai, theo đợt kiểm tra trong dòng 1 và Bổ đề 27.28, ta chỉ cần chứng tỏ tất cả các cạnh rời u là không chấp nhận được. Nhận thấy khi DISCHARGE(u) được gọi liên tục, biến trỏ $current[u]$ dời xuống danh sách $N[u]$. Mỗi "lượt" bắt đầu tại đầu của $N[u]$ và hoàn tất với $current[u]$ = NIL, tại đó u được nâng và một lượt mới bắt đầu. Để biến trỏ $current[u]$ tiến tới vượt quá một đỉnh $v \in N[u]$ trong một lượt, cạnh (u, v) phải được đợt kiểm tra trong dòng 6 nghĩ là không chấp nhận được. Như vậy, vào lúc lượt đó hoàn tất, mọi cạnh rời u đã được xác định là không chấp nhận được vào một thời điểm nào đó trong lượt đó. Nhận xét chính đó là vào cuối lượt, mọi cạnh rời u vẫn không chấp nhận được. Tại sao? Theo Bổ đề 27.27, phép đẩy không thể tạo các cạnh chấp nhận được, để mỗi một cạnh rời u. Như vậy, các cạnh chấp nhận được phải được tạo bởi một phép nâng. Nhưng đỉnh u không được nâng trong lượt đó, và theo Bổ đề 27.28, mọi đỉnh v khác

được nâng trong lượt không có các cạnh chấp nhận được nhập vào. Như vậy, vào cuối lượt, tất cả các cạnh rời u đều vẫn là không chấp nhận được, và bổ đề được chứng minh.

Thuật toán nâng tới trước

Trong thuật toán nâng tới trước, ta duy trì một danh sách nối kết L bao gồm tất cả các đỉnh trong $V - \{s, t\}$. Một tính chất chính đó là các đỉnh trong L được sắp xếp tôpô theo mạng chấp nhận được. (Hãy nhớ lại Bổ đề 27.26, mạng chấp nhận được là một dag.)

Mã giả của thuật toán nâng tới trước mặc nhận các danh sách láng giềng $N[u]$ đã được tạo cho mỗi đỉnh u. Nó cũng mặc nhận *next*[u] trỏ đến đỉnh theo sau u trong danh sách L và, như thường lệ, *next*[u] = NIL nếu u là đỉnh cuối trong danh sách.

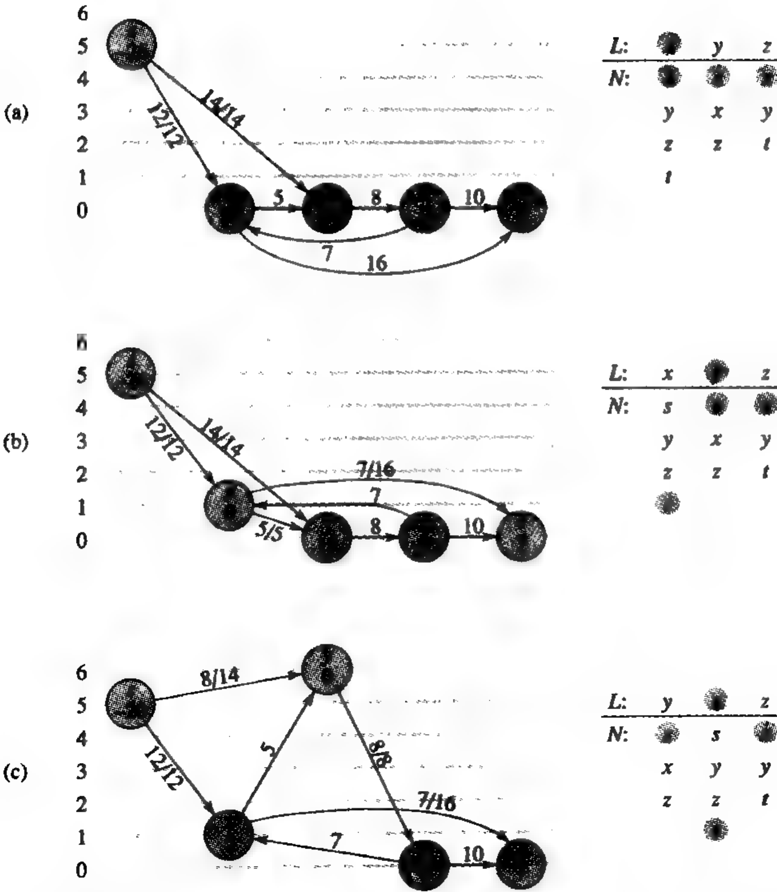
```
LIFT-TO-FRONT(G, s, t)
1   INITIALIZE-PREFLOW(G, s)
2   L ← V[G] − {s, t}, theo một thứ tự bất kỳ
3   for mỗi đỉnh u ∈ V[G] − {s, t}
4       do current[u] ← head[N[u]]
5   u ← head[L]
6   while u ≠ NIL
7       do old-height ← h[u]
8          DISCHARGE(u)
9          if h[u] > old-height
10            then dời u các đầu danh sách L
11         u ← next[u]
```

Thuật toán nâng tới trước làm việc như sau. Dòng 1 khởi tạo luồng trước và các chiều cao theo cùng các giá trị như trong thuật toán đẩy luồng trước chung. Dòng 2 khởi tạo danh sách L để chứa tất cả các đỉnh có tiềm năng tràn, theo một thứ tự bất kỳ. Các dòng 3-4 khởi tạo biến trỏ *current* của mỗi đỉnh u theo đỉnh đầu tiên trong danh sách láng giềng của u.

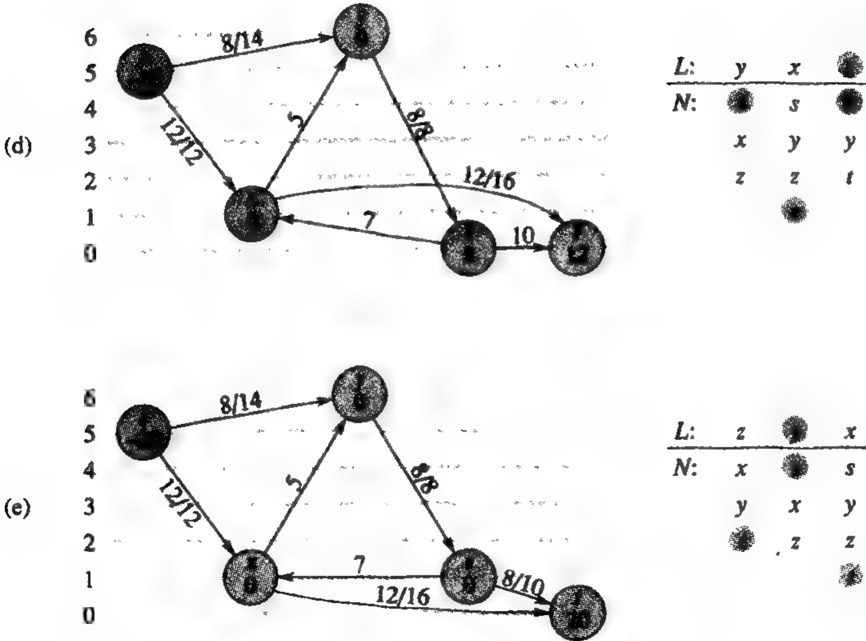
Như đã nêu trong Hình 27.11, vòng lặp **while** của các dòng 6-11 chạy qua danh sách L, xả các đỉnh. Dòng 5 khiến nó bắt đầu với đỉnh đầu tiên trong danh sách. Mỗi lần qua vòng lặp, một đỉnh u được xả

trong dòng 8. Nếu *u* được nâng bởi thủ tục DISCHARGE, dòng 10 dời nó đến đầu danh sách *L*. Phép xác định này được thực hiện bằng cách lưu chiều cao của *u* trong biến *old-height* trước phép toán xả (dòng 7) và so sánh chiều cao đã lưu này với chiều cao của *u* sau đó (dòng 9). Dòng 11 khiến lần lặp lại kế tiếp của vòng lặp **while** dùng đỉnh theo sau *u* trong danh sách *L*. Nếu *u* đã được dời đến đầu danh sách, đỉnh được dùng trong lần lặp lại kế tiếp sẽ là đỉnh theo sau *u* tại vị trí mới của nó trong danh sách.

Để chứng tỏ LIFT-TO-FRONT tính toán một luồng cực đại, ta sẽ chứng tỏ nó là một thực thi của thuật toán đẩy luồng trước chung. Trước tiên, nhận thấy nó chỉ thực hiện phép đẩy và nâng khi chúng được áp dụng, bởi Bổ đề 27.29 bảo đảm DISCHARGE chỉ thực hiện chúng khi chúng được áp dụng. Ta còn phải chứng tỏ khi LIFT-TO-FRONT kết thúc, không có phép toán cơ bản nào được áp dụng. Nhận thấy nếu *u* đụng cuối *L*, mọi đỉnh trong *L* phải được xả mà không tạo ra một phép nâng. Bổ đề 27.30, mà ta sẽ chứng minh dưới đây, phát biểu rằng danh sách *L* được duy trì dưới dạng một đợt sắp xếp tôpô của mạng chấp nhận được. Như vậy, một phép đẩy sẽ khiến luồng thặng dư dời đến các đỉnh xa hơn xuống dưới danh sách (hoặc đến *s* hay *t*). Do đó, nếu biến trỏ *u* đụng cuối danh sách, phần dư của mọi đỉnh là 0, và không có phép toán cơ bản nào được áp dụng.



Hình 27.11 Hành động của LIFT-TO-FRONT. **(a)** Một mạng luồng ngay trước lần lặp lại đầu tiên của vòng lặp **while**. Thoạt đầu, 26 đơn vị của luồng rời nguồn *s*. Bên phải nêu danh sách ban đầu $L = <x, y, z>$, ở đó thoạt đầu $u = x$. Dưới mỗi đỉnh trong danh sách L là danh sách láng giềng của nó, với láng giềng hiện hành được tô bóng. Đỉnh *x* được xả. Nó được nâng lên chiều cao 1, 5 đơn vị của luồng thặng dư được đẩy đến *y*, và 7 đơn vị còn lại của phần thặng dư được đẩy đến bồn *t*. Bởi *x* được nâng, nên nó được dời đến đầu của *L*, mà trong trường hợp này không thay đổi cấu trúc của *L*. **(b)** Sau *x*, đỉnh kế tiếp trong *L* được xả là *y*. Hình 27.10 nêu hành động chi tiết của tiến trình xả *y* trong tình huống này. Bởi *y* được nâng, nên nó được dời đến đầu của *L*. **(c)** Đỉnh *x* giờ đây theo *y* trong *L*, và do đó nó lại được xả, đẩy tất cả 5 đơn vị của luồng thặng dư đến *t*. Bởi đỉnh *x* không được nâng trong phép toán xả này, nên nó vẫn giữ nguyên tại chỗ trong danh sách *L*. **(d)** Bởi đỉnh *z* theo đỉnh *x* trong *L*, nên nó được xả. Nó được nâng lên chiều cao 1 và tất cả 8 đơn vị của luồng thặng dư được đẩy đến *t*. Bởi *z* được nâng, nên nó được dời đến đầu *L*. **(e)** Đỉnh *y* giờ đây theo đỉnh *z* trong *L* và do đó được xả. Nhưng do *y* không có phần thặng dư, nên DISCHARGE lập tức trở về, và *y* vẫn giữ nguyên tại chỗ trong *L*. Như vậy, đỉnh *x* được xả. Bởi nó cũng không có phần thặng dư, DISCHARGE lại trở về, và *x* giữ nguyên tại chỗ trong *L*. LIFT-TO-FRONT đã đụng cuối danh sách *L* và kết thúc. Không có các đỉnh tràn, và luồng trước là một luồng cực đại.



Bổ đề 27.30

Nếu ta chạy LIFT-TO-FRONT trên một mạng luồng $G' = (V, E)$ có nguồn s và bồn t, thì mỗi lần lặp lại của vòng lặp **while** trong các dòng 6-11 sẽ duy trì sự bất biến rằng danh sách L là một đợt sắp xếp tôpô của các đỉnh trong mạng chấp nhận được $G_{f,h} = (V, E_{f,h})$.

Chứng minh Lập tức sau khi INITIALIZE-PREFLOW đã chạy, $h[s] = |V|$ và $h[v] = 0$ với tất cả $v \in V - \{s\}$. Bởi $|V| \geq 2$ (bởi nó chứa ít nhất s và t), nên không có cạnh nào có thể chấp nhận được. Như vậy, $E_{f,h} = \emptyset$, và mọi kiểu sắp xếp của $V - \{s, t\}$ đều là một đợt sắp xếp tôpô của $G_{f,h}$.

Giờ đây ta chứng tỏ sự bất biến được duy trì bởi mỗi lần lặp lại của vòng lặp **while**. Mạng chấp nhận được chỉ thay đổi bằng các phép đẩy và nâng. Theo Bổ đề 27.27, các phép toán đẩy chỉ tạo các cạnh không chấp nhận được. Như vậy, các cạnh chấp nhận được chỉ có thể được tạo bởi các phép nâng. Tuy nhiên, sau khi một đỉnh được nâng, Bổ đề 27.28 phát biểu rằng không có các cạnh chấp nhận được nhập u nhưng có thể có các cạnh chấp nhận được rời u. Như vậy, khi dời u đến đầu L, thuật toán bảo đảm mọi cạnh chấp nhận được rời u đều thỏa kiểu sắp xếp tôpô.

Phân tích

Giờ đây ta chứng tỏ LIFT-TO-FRONT chạy trong $O(V^3)$ thời gian

trên một mạng luồng $G = (V, E)$. Bởi thuật toán là một thực thi của thuật toán đẩy luồng trước chung, ta sẽ vận dụng Hệ luận 27.21, cung cấp một cận $O(V)$ trên số lượng các phép nâng thi hành cho mỗi đỉnh và một cận $O(V^2)$ trên tổng các phép nâng nói chung. Ngoài ra, Bài tập 27.4-2 cung cấp một cận $O(VE)$ trên tổng thời gian bỏ ra để thực hiện các phép nâng, và Bổ đề 27.22 cung cấp một cận $O(VE)$ trên tổng các phép đẩy bão hòa.

Định lý 27.31

Thời gian thực hiện của LIFT-TO-FRONT trên một mạng luồng $G = (V, E)$ là $O(V^3)$.

Chứng minh Ta hãy xét một "giai đoạn" của thuật toán nâng tới trước là thời gian giữa hai phép nâng liên tiếp. Có $O(V^2)$ giai đoạn, bởi có $O(V^2)$ phép nâng. Mỗi giai đoạn bao gồm tối đa $|V|$ lệnh gọi DISCHARGE, có thể được xem xét như sau. Nếu DISCHARGE không thực hiện một phép nâng, lệnh gọi kế tiếp đến DISCHARGE sẽ hạ xuống thêm trong danh sách L, và chiều dài của L sẽ nhỏ hơn $|V|$. Nếu DISCHARGE thực hiện một phép nâng, lệnh gọi kế tiếp đến DISCHARGE thuộc về một giai đoạn khác. Bởi mỗi giai đoạn chứa tối đa $|v|$ lệnh gọi đến DISCHARGE và có $O(V^2)$ giai đoạn, số lần DISCHARGE được gọi trong dòng 8 của LIFT-TO-FRONT là $O(V^3)$. Như vậy, ngoại trừ công việc được thực hiện trong DISCHARGE, tổng công việc được thực hiện bởi vòng lặp **while** trong LIFT-TO-FRONT tối đa là $O(V^3)$.

Giờ đây ta phải định cận công việc được thực hiện trong DISCHARGE trong khi thi hành thuật toán. Mỗi lần lặp lại của vòng lặp **while** trong DISCHARGE thực hiện một trong ba hành động. Ta sẽ phân tích tổng lượng công việc có liên quan trong khi thực hiện từng ngày này.

Ta bắt đầu với các phép nâng (các dòng 4-5). Bài tập 27.4-2 cung cấp một cận thời gian $O(VE)$ trên tất cả $O(V^2)$ phép nâng được thực hiện.

Giờ đây, giả sử rằng hành động cập nhật biến trỏ *current*[*u*] trong dòng 8. Hành động này xảy ra $O(\text{độ}(u))$ lần mỗi lần có một đỉnh u được nâng, và $O(V \cdot \text{degree}(u))$ lần chung cho đỉnh đó. Do đó, với tất cả các đỉnh, tổng lượng công việc thực hiện trong việc đưa các biến trỏ ra phía trước trong các danh sách láng giềng là $O(VE)$ theo bổ đề thiết lập quan hệ (Bài tập 5.4-1).

Kiểu hành động thứ ba được DISCHARGE thực hiện đó là một phép đẩy (dòng 7). Giờ đây ta đã biết tổng các phép đẩy bão hòa là $O(VE)$. Nhận thấy nếu một phép đẩy không bão hòa được thi hành, DISCHARGE lập tức trở về, bởi phép đẩy rút gọn phần thặng dư thành 0. Như vậy, có thể có tối đa một phép đẩy không bão hòa cho mỗi lệnh gọi DISCHARGE.

Như đã nhận xét, DISCHARGE được gọi $O(V^3)$ lần, và như vậy tổng thời gian bỏ ra để thực hiện các phép đẩy không bão hòa là $O(V^3)$.

Do đó, thời gian thực hiện của LIFT-TO-FRONT là $O(V^3 + VE)$, chính là $O(V^3)$.

Bài tập

27.5-1

Minh họa tiến trình thi hành của LIFT-TO-FRONT theo cách trong Hình 27.11 với mạng luồng trong Hình 27.1(a). Giả sử cách sắp xếp ban đầu của các đỉnh trong L là $<v_1, v_2, v_3, v_4>$ và các danh sách láng giềng là

$N[v_1] = <s, v_2, v_3>$,

$N[v_2] = <s, v_1, v_3, v_4>$,

$N[v_3] = <v_1, v_2, v_4, t>$,

$N[v_4] = <v_2, v_3, t>$.

*27.5-2 **

Ta muốn thực thi một thuật toán đẩy luồng trước ở đó duy trì một hàng đợi vào trước ra trước gồm các đỉnh tràn. Thuật toán liên tục xả đỉnh tại đầu hàng đợi; và mọi đỉnh không tràn trước lần xả nhưng tràn sau đó sẽ được đặt tại cuối hàng đợi. Sau khi xả đỉnh tại đầu hàng đợi, nó được gỡ bỏ. Khi hàng đợi trống, thuật toán kết thúc. Chứng tỏ thuật toán này có thể thực thi để tính toán một luồng cực đại trong $O(V^3)$ thời gian.

27.5-3

Chứng tỏ thuật toán chung vẫn làm việc nếu LIFT cập nhật $h[u]$ bằng cách đơn giản tính toán $h[u] \leftarrow h[u] + 1$. Thay đổi này ảnh hưởng như thế nào đến phần phân tích của LIFT-TO-FRONT

*27.5-4 **

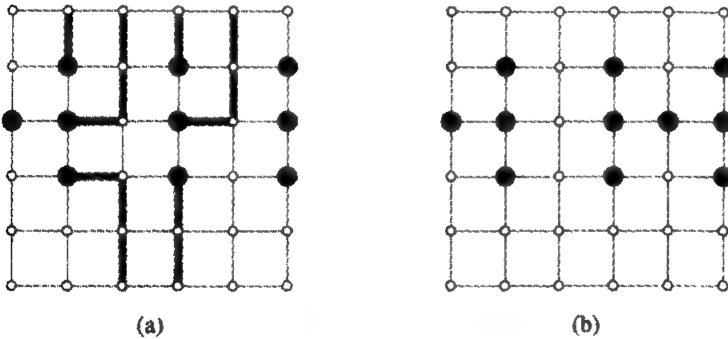
Chứng tỏ nếu luôn xả một đỉnh tràn cao nhất, ta có thể thực hiện phương pháp đẩy luồng trước để chạy trong $O(V^3)$ thời gian.

Các Bài Toán

27-1 Bài toán lối thoát

Một khung kẻ ô $n \times n$ là một đồ thị không hướng bao gồm n hàng và

n cột gồm các đỉnh, như đã nêu trong Hình 27.12. Ta thể hiện đỉnh trong hàng thứ *i* và cột thứ *j* bằng (i, j). Tất cả các đỉnh trong một khung kẻ ô có chính xác bốn láng giềng, ngoại trừ các đỉnh biên, là các điểm (i, j) mà $i = 1, i = n, j = 1$, hoặc $j = n$.



Hình 27.12 Các khung kẻ ô cho bài toán lối thoát. Các điểm bắt đầu được tô đen, và các đỉnh khác được tô trắng. **(a)** Một khung kẻ ô có một lối thoát, được nêu bằng các lộ trình tô bóng. **(b)** Một khung kẻ ô không có lối thoát.

Cho $m \leq n^2$ điểm bắt đầu $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$ trong khung kẻ ô, ***bài toán lối thoát*** đó là xác định có hay không *m* lộ trình đỉnh rời nhau từ điểm bắt đầu đến bất kỳ *m* điểm khác nhau trên biên. Ví dụ, khung kẻ ô trong Hình 27.12(a) có một lối thoát, nhưng khung kẻ ô trong Hình 27.12(b) thì không.

a. Xét một mạng luồng ở đó các đỉnh, cũng như các cạnh, có các dung lượng. Nghĩa là, luồng mạng dương nhập bất kỳ đỉnh đã cho nào đều tùy thuộc vào một ràng buộc dung lượng. Chứng tỏ có thể rút gọn việc xác định luồng cực đại trong một mạng có các dung lượng cạnh và đỉnh theo một bài toán luồng cực đại bình thường trên một mạng luồng có quy mô so sánh được.

b. Mô tả một thuật toán hiệu quả để giải quyết bài toán lối thoát, và phân tích thời gian thực hiện của nó.

27-2 Vỏ bọc lộ trình cực tiểu

Một ***vỏ phủ lộ trình*** [path cover] của một đồ thị có hướng $G = (V, E)$ là một tập hợp *P* lộ trình đỉnh rời nhau sao cho mọi đỉnh trong *V* nằm trong chính xác một lộ trình trong *P*. Các lộ trình có thể bắt đầu và kết thúc bất kỳ đâu, và chúng có thể có bất kỳ chiều dài nào, kể cả 0. Một ***vỏ phủ lộ trình cực tiểu*** của *G* là một vỏ phủ lộ trình chứa ít lộ trình khả dĩ nhất.

a. Nêu một thuật toán hiệu quả để tìm một vỏ phủ lộ trình cực tiểu của một đồ thị phi chu trình có hướng $G = (V, E)$. (*Mách nước:* Giả sử *V*

= {1, 2,..., n}, kiến tạo đồ thị $G' = (V', E')$, ở đó

$V' = \{x_0, x_1, ..., x_n\} \cup \{y_0, y_1, ..., y_n\}$

$E' = \{(x_0, x_i) : i \in V\} \cup \{(y_i, y_0) : i \in V\} \cup \{(x_i, y_j) : (i, j) \in E\}$

và chạy một thuật toán luồng cực đại.)

b. Thuật toán của bạn có làm việc cho đồ thị có hướng chứa các chu trình không? Giải thích.

27-3 Các thí nghiệm con thoi không gian

Giáo sư Spock đang cố vấn cho NASA, cơ quan này đang hoạch định một loạt các chuyến bay con thoi không gian và phải quyết định các cuộc thí nghiệm thương mại để thực hiện và các dụng cụ phải có trên tàu cho mỗi chuyến bay. Với mỗi chuyến bay, NASA xét một tập hợp $E = \{E_1, E_2,..., E_m\}$ thí nghiệm, và nhà bảo trợ thương mại cho thí nghiệm E_j đã đồng ý thanh toán cho NASA p_j đôla cho các kết quả của thí nghiệm. Các cuộc thí nghiệm dùng một tập hợp $I = \{I_1, I_2,..., I_n\}$ dụng cụ; mỗi thí nghiệm E_j yêu cầu tất cả các dụng cụ trong một tập hợp con $R_j \subseteq I$. Mức hao phí để mang dụng cụ I_k là c_k đôla. Công việc của giáo sư đó là tìm một thuật toán hiệu quả để xác định các cuộc thí nghiệm để thực hiện và các dụng cụ để mang cho một chuyến bay đã cho để tối đa hóa lợi tức ròng, là tổng thu nhập từ các cuộc thí nghiệm được thực hiện trừ tổng mức hao phí của tất cả dụng cụ mang theo.

Xét mạng G dưới đây. Mạng chứa một đỉnh nguồn s, các đỉnh I_1, I_2,..., I_n, các đỉnh E_1, E_2,..., E_m, và một bồn đỉnh t. Với k = 1, 2..., n, ta có một cạnh (s, I_k) có dung lượng c_k, và với j = 1, 2,..., m, ta có một cạnh (E_j, t) có dung lượng p_j. Với k = 1, 2,..., n và j = 1, 2,..., m, nếu $I_k \in R_j$, thì có một cạnh (I_k, E_j) có dung lượng vô hạn.

a. Chứng tỏ nếu $E_j \in T$ với một phần cắt có dung lượng hữu hạn (S, T) của G, thì $I_k \in T$ với mỗi $I_k \in R_j$.

b. Nêu cách xác định lợi tức ròng cực đại từ dung lượng của phần cắt cực tiểu của G và p_j giá trị đã cho.

c. Nêu một thuật toán hiệu quả để xác định các cuộc thí nghiệm để thực hiện và các dụng cụ mang theo. Phân tích thời gian thực hiện của thuật toán theo dạng m, n, và $r = \Sigma_{j=1}^{m} |R_j|$

27-4 Cập nhật luồng cực đại

Cho $G = (V, E)$ là một mạng luồng có nguồn s, bồn t, và các dung lượng số nguyên. Giả sử ta có một luồng cực đại trong G.

a. Giả sử dung lượng của một cạnh đơn $(u, v) \in E$ tăng theo 1. Nêu một thuật toán $O(V + E)$ thời gian để cập nhật luồng cực đại.

b. Giả sử dung lượng của một cạnh đơn $(u, v) \in$ E giảm theo 1. Nêu một thuật toán $O(V +E)$ thời gian để cập nhật luồng cực đại.

27-5 Luồng cực đại bằng định tỷ lệ

Cho $G = (V, E)$ là một mạng luồng có nguồn s, bồn t, và một dung lượng số nguyên $c(u, v)$ trên mỗi cạnh $(u, v) \in E$. Cho $C = \max_{(u,v)\in E} c(u, v)$.

a. Chứng tỏ một phần cắt cực tiểu của G có dung lượng tối đa $C|E|$.

b. Với một số K đã cho, chứng tỏ có thể tìm thấy một lộ trình tăng cường có dung lượng ít nhất là K trong $O(E)$ thời gian, nếu như có một lộ trình như vậy tồn tại.

Có thể dùng phần sửa đổi dưới đây của FORD-FULKERSON-METHOD để tính toán một luồng cực đại trong G.

MAX-FLOW-BY-SCALING(G, s, t)

1 $C \leftarrow \max_{(u,v)\in E} c(u, v)$

2 khởi tạo luồng f theo 0

3 $K \in 2^{\lfloor \lg C \rfloor}$

4 **while** $K \geq 1$

5 **do while** ở đó tồn tại một lộ trình tăng cường p có dung lượng ít nhất là K

6 **do** tăng cường luồng f dọc theo p

7 $K \leftarrow K/2$

8 **return** f

c. Chứng tỏ MAX-FLOW-BY-SCALING trả về một luồng cực đại.

d. Chứng tỏ dung lượng thặng dư của một phần cắt cực tiểu của G tối đa là $2K|E|$ mỗi lần dòng 4 được thi hành.

e. Chứng tỏ vòng lặp **while** phía trong của các dòng 5-6 được thi hành $O(E)$ lần với mỗi giá trị của K.

f. Kết luận rằng MAX-FLOW-BY-SCALING có thể được thực thi để chạy trong $O(E^2 \lg C)$ thời gian.

27-6 Luồng cực đại với các cận dung lượng cao và thấp hơn

Giả sử mỗi cạnh (u, v) trong một mạng luồng $G = (V, E)$ không

những có một cận trên $c(u, v)$ trên luồng mạng từ u đến v, mà còn có một cận dưới $b(u, v)$. Nghĩa là, một luồng f trên mạng phải thỏa $b(u, v) \leq f(u, v) \leq c(u, v)$. Có thể đó là trường hợp của một mạng ở đó không tồn tại luồng khả thi nào.

a. Chứng minh nếu f là một luồng trên G, thì $|f| \leq c(S, T) - b(T, S)$ với bất kỳ phần cắt (S, T) nào của G.

b. Chứng minh giá trị của một luồng cực đại trong mạng, nếu nó tồn tại, là giá trị cực tiểu của $c(S, T) - b(T, S)$ trên tất cả các phần cắt (S, T) của mạng.

Cho $G = (V, E)$ là một mạng luồng có các hàm cận trên và dưới c và b, và cho s và t là nguồn và bồn của G. Kiến tạo mạng luồng bình thường $G' = (V', E')$ với hàm cận trên c', nguồn s', và bồn t' như sau:

$V' = V \cup \{s', t'\}$,

$E' = E \cup \{(s', v) : v \in V\} \cup \{(u, t') : u \in V\} \cup \{(s, t), (t, s)\}$.

Ta gán các dung lượng cho các cạnh như sau. Với mỗi cạnh $(u, v) \in E$, ta ấn định $c'(u, v) = c(u, v) - b(u, v)$. Với mỗi đỉnh $u \in V$, ta ấn định $c'(s', u) = b(V, u)$ và $c'(u, t') = b(u, V)$. Ta cũng ấn định $c'(s, t) = c'(t, s) = \infty$.

c. Chứng minh ở đó tồn tại một luồng khả thi trong G nếu và chỉ nếu tồn tại một luồng cực đại trong G' sao cho tất cả các cạnh vào bồn t' đều bão hòa.

d. Nêu một thuật toán tìm một luồng cực đại trong một mạng có các cận trên và dưới hoặc xác định không có luồng khả thi nào tồn tại. Phân tích thời gian thực hiện của thuật toán.

Ghi chú Chương

Even [65], Lawler [132], Papadimitriou và Steiglitz [154], và Tarjan [188] là những tham chiếu tốt về luồng mạng và các thuật toán có liên quan. Goldberg, Tardos, và Tarjan [83] cung cấp một nghiên cứu thú vị về các thuật toán cho các bài toán luồng mạng.

Phương pháp Ford-Fulkerson là do Ford và Fulkerson [71], họ đã phát sinh nhiều bài toán trong lĩnh vực luồng mạng, kể cả các bài toán luồng cực đại và so khớp hai nhánh. Nhiều thực thi tiên khởi của phương pháp Ford-Fulkerson đã thấy các lộ trình tăng cường dùng thuật toán tìm kiếm độ rộng đầu tiên; Edmonds và Karp [63] đã chứng minh chiến lược này cho ra một thuật toán thời gian đa thức. Karzanov [119] đã

phát triển ý tưởng các luồng trước. Phương pháp đẩy luồng trước là của Goldberg [82]. Thuật toán đẩy luồng trước nhanh nhất ngày nay là của Goldberg và Tarjan [85], họ đạt được một thời gian thực hiện là $O(VE \lg(V^2/E))$. Thuật toán tốt nhất hiện nay về so khớp hai nhánh cực đại, đã được khám phá bởi Hopcroft và Karp [10 1], chạy trong $O(\sqrt{V} E)$ thời gian.

VII Các Chủ Đề Chọn Lọc

Mở đầu

Phần này bao gồm sự lựa chọn các chủ đề thuật toán mở rộng và bổ trợ cho nội dung trước đây trong cuốn sách này. Có vài chương giới thiệu các mô hình mới về tính toán như các mạch tổ hợp hoặc các máy tính song song. Các chương khác đề cập các lĩnh vực chuyên môn như hình học tính toán hoặc lý thuyết số. Hai chương cuối đề cập vài giới hạn đã biết để thiết kế các thuật toán hiệu quả và giới thiệu các kỹ thuật để đối phó với các giới hạn đó.

Chương 28 trình bày mô hình tính toán song song đầu tiên của chúng ta: các mạng so sánh. Đại khái, một mạng so sánh là một thuật toán cho phép thực hiện nhiều phép so sánh đồng thời. Chương này nêu cách xây dựng một mạng so sánh có thể sắp xếp n con số trong $O(\lg^2 n)$ thời gian.

Chương 29 giới thiệu một mô hình tính toán song song khác: các mạch tổ hợp. Chương này chứng tỏ hai số n-bit có thể được bổ sung trong $O(\lg n)$ thời gian dùng một mạch tổ hợp có tên bộ cộng nhớ kiểm tra trước. Nó cũng nêu cách nhân hai số n-bit trong $O(\lg n)$ thời gian.

Chương 30 giới thiệu một mô hình tính toán song song chung có tên PRAM. Chương trình bày các kỹ thuật song song cơ bản, bao gồm kỹ thuật nhảy biến trỏ, các phép tính tiền tố, và kỹ thuật tua Euler. Hầu hết các kỹ thuật được minh họa dựa trên các cấu trúc dữ liệu đơn giản, kể cả các danh sách và các cây. Chương cũng mô tả các vấn đề chung trong tính toán song song, kể cả tính hiệu quả công [work efficiency] và truy cập đồng thời vào bộ nhớ dùng chung. Nó chứng minh định lý Brent, nêu cách thức mà một máy tính song song có thể mô phỏng hiệu quả một mạch tổ hợp. Chương kết thúc với một thuật toán ngẫu nhiên hóa hiệu quả công để xếp loại danh sách và một thuật toán tất định hiệu quả đáng kể để ngắt tính đối xứng trong một danh sách.

Chương 31 nghiên cứu các thuật toán hiệu quả để vận hành trên các ma trận. Nó bắt đầu bằng thuật toán Strassen, có thể nhân hai ma trận

$n \times n$ trong $O(n^{2.81})$ thời gian. Sau đó trình bày hai phương pháp chung—phân tích LU và phân tích LUP—để giải các phương trình tuyến tính bằng phép khử Gauss trong $O(n^3)$ thời gian. Nó cũng chứng tỏ có thể dùng thuật toán Strassen để giải tuyến tính các hệ thống nhanh hơn và chứng tỏ có thể thực hiện phép nghịch đảo ma trận và phép nhân ma trận nhanh bằng nhau, theo tiệm cận. Chương kết thúc với phần trình bày cách tính toán giải pháp xấp xỉ các bình phương bé nhất [least-squares] khi một tập hợp các phương trình tuyến tính không có giải pháp chính xác.

Chương 32 nghiên cứu các phép toán trên các đa thức và chứng tỏ một kỹ thuật xử lý tín hiệu nổi tiếng—Fast Fourier Transform (FFT)—có thể được dùng để nhân hai đa thức bậc-n trong $O(n \lg n)$ thời gian. Nó cũng nghiên cứu các kiểu thực thi hiệu quả của FFT, kể cả mạch song song.

Chương 33 trình bày các thuật toán lý thuyết số. Sau khi ôn lại lý thuyết số cơ bản, nó trình bày thuật toán Euclid để tính toán các số chia chung lớn nhất. Chương cũng trình bày các thuật toán để giải các phương trình tuyến tính mođun và để nâng một số lên một lũy thừa modulo một số khác. Sau đó, trình bày một ứng dụng đáng quan tâm của các thuật toán lý thuyết số: hệ mật mã khóa công RSA. Hệ mật mã này không những có thể dùng để mã hóa các thông điệp sao cho đối phương không thể đọc chúng, nó còn được dùng để cung cấp các ký danh số hóa [digital signatures]. Sau đó, ta sẽ đề cập đợt thử tính nguyên thủy Miller-Rabin ngẫu nhiên hóa, có thể dùng để tìm các số nguyên tố lớn một cách hiệu quả—một yêu cầu thiết yếu của hệ thống RSA. Cuối cùng, ta đề cập heuristic "rho" của Pollard để phân tích thành thừa số các số nguyên và mô tả nét ưu việt của phép tìm thừa số số nguyên.

Chương 34 nghiên cứu bài toán tìm tất cả các lần xuất hiện của một khuôn mẫu chuỗi đã cho trong một văn bản chuỗi đã cho, một bài toán thường gặp trong các chương trình hiệu chỉnh văn bản. Cách tiếp cận lịch lãm của Rabin và Karp sẽ được xem xét đầu tiên. Và sau khi xét một giải pháp hiệu quả dựa trên otomat hữu hạn, ta sẽ mô tả thuật toán Knuth-Morris-Pratt, đạt hiệu năng bằng cách xử lý trước khuôn mẫu một cách thông minh. Chương kết thúc với một phần trình bày về một heuristic so khớp chuỗi của Boyer và Moore.

Hình học tính toán là chủ đề của Chương 35. Sau khi mô tả các nguyên hàm [primitives] cơ bản của hình học tính toán, chương sẽ nêu cách thức mà phương pháp "quét" có thể xác định một cách hiệu quả xem một tập hợp các đoạn thẳng có chứa giao điểm nào không. Hai thuật toán thông minh để tìm bao lồi [convex hull] của một tập hợp các

điểm—quét Graham và diễn tiến Jarvis—cũng minh họa năng lực của các phương pháp quét. Chương kết thúc bằng một thuật toán hiệu quả để tìm cặp sát nhất trong số một tập hợp các điểm đã cho trong mặt phẳng.

Chương 36 đề cập các bài toán đầy đủ NP. Nhiều bài toán tính toán đáng quan tâm mang tính đầy đủ NP, nhưng hiện chưa có thuật toán thời gian đa thức nào để giải chúng. Chương này trình bày các kỹ thuật để xác định thời điểm mà một bài toán mang tính đầy đủ NP. Một số bài toán cổ điển được chứng minh là đầy đủ NP: xác định xem một đồ thị có một chu trình hamilton hay không, xác định xem một công thức Bool có thỏa đáng không, và xác định xem một tập hợp các con số đã cho có một tập hợp con cộng dồn đến một giá trị đích đã cho hay không.

Chương cũng chứng minh bài toán người bán hàng du hành nổi tiếng có đầy đủ NP.

Chương 37 nêu cách dùng các thuật toán xấp xỉ để tìm các giải pháp xấp xỉ cho các bài toán đầy đủ NP một cách hiệu quả. Với vài bài toán đầy đủ NP, các giải pháp xấp xỉ gần tối ưu thường không khó tạo, nhưng với các giải pháp khác kể cả các thuật toán xấp xỉ tốt nhất mà ta được biết làm việc tệ dần khi kích cỡ bài toán gia tăng. Ngoài ra, có vài bài toán mà ta có thể đầu tư các lượng thời gian tính toán tăng để đổi lại có được các giải pháp xấp xỉ tốt dần lên. Chương này minh họa các khả năng này bằng bài toán vỏ phủ đỉnh, bài toán người bán hàng du hành, bài toán phủ tập hợp, và bài toán tổng tập hợp con.

28 Các Mạng Sắp Xếp

Trong Phần II, ta đã xem xét các thuật toán sắp xếp cho các máy tính nối tiếp (các máy truy cập ngẫu nhiên, hoặc các RAM) chỉ cho phép lần lượt thi hành từng phép toán một. Trong chương này, ta nghiên cứu các thuật toán sắp xếp dựa trên một mô hình tính toán mạng so sánh ở đó có thể thực hiện nhiều phép so sánh đồng thời.

Các mạng so sánh khác với các RAM ở hai điểm quan trọng. Thứ nhất, chúng chỉ có thể thực hiện các phép so sánh. Như vậy, ta không thể thực thi một thuật toán như sắp xếp đếm (xem Đoạn 9.2) trên một mạng so sánh. Thứ hai, khác với mô hình RAM, ở đó các phép toán xảy ra tuần tự—nghĩa là, cái này sau cái kia—các phép toán trong một mạng so sánh có thể xảy ra cùng một lúc, hoặc "song song." Như sẽ thấy, đặc tính này cho phép kiến tạo các mạng so sánh sắp xếp n giá trị trong thời gian tuyến tính con [sublinear].

Ta bắt đầu Đoạn 28.1 bằng cách định nghĩa các mạng so sánh và các mạng sắp xếp. Ta cũng cung cấp một định nghĩa tự nhiên về "thời gian thực hiện" của một mạng so sánh theo dạng độ sâu của mạng. Đoạn 28.2 chứng minh "nguyên lý zero-một," tạo thuận lợi rất nhiều cho việc phân tích tính đúng đắn của các mạng sắp xếp.

Mạng sắp xếp hiệu quả mà ta thiết kế chủ yếu là một phiên bản song song của thuật toán sắp xếp trộn trong Đoạn 1.3.1. Phần kiến tạo của chúng ta gồm có ba bước. Đoạn 28.3 trình bày thiết kế của một bộ sắp xếp "bitonic" sẽ là khối kiến tạo cơ bản của chúng ta. Ta sửa đổi chút đỉnh bộ sắp xếp bitonic trong Đoạn 28.4 để tạo một mạng trộn [merging network] có thể trộn hai dãy sắp xếp thành một dãy sắp xếp. Cuối cùng, trong Đoạn 28.5, ta gom các mạng trộn này thành một mạng sắp xếp có thể sắp xếp n giá trị trong $O(\lg^2 n)$ thời gian.

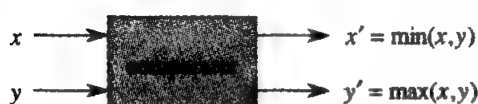
28.1 Các mạng so sánh

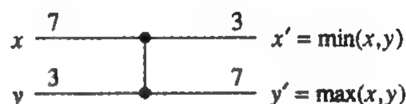
Các mạng sắp xếp là những mạng so sánh luôn sắp xếp các đầu vào của chúng, do đó tốt nhất ta nên bắt đầu bằng các mạng so sánh và các đặc tính của chúng. Một mạng so sánh chỉ gồm có các dây dẫn và các

bộ so sánh. Một ***bộ so sánh***, xem Hình 28.1(a), là một thiết bị có hai phần nhập, x và y, và hai phần xuất, x' và y', thực hiện hàm dưới đây:

$$x' = \min(x,y)$$

$$y' = \max(x,y)$$





Hình 28.1 (a) Một bộ so sánh có các phần nhập x và y và các phần xuất x' và y'. **(b)** Cùng bộ so sánh, được vẽ dưới dạng một vạch dọc đơn. Các phần nhập $x = 7$, $y = 3$ và các phần xuất $x' = 3$, $y' = 7$ được nêu.

Do phần biểu diễn hình ảnh của bộ so sánh trong Hình 28.1(a) quá kềnh càng đối với các mục tiêu của chúng ta, nên ta sẽ chấp nhận quy ước vẽ các bộ so sánh dưới dạng các vạch dọc đơn, như đã nêu trong Hình 28.1(b). Các phần nhập xuất hiện bên trái và các phần xuất ở bên phải, với giá trị nhập nhỏ hơn xuất hiện trên kết xuất đỉnh và giá trị nhập lớn hơn xuất hiện trên kết xuất đáy. Như vậy, ta có thể xem bộ so sánh như một tiến trình sắp xếp hai đầu vào của nó.

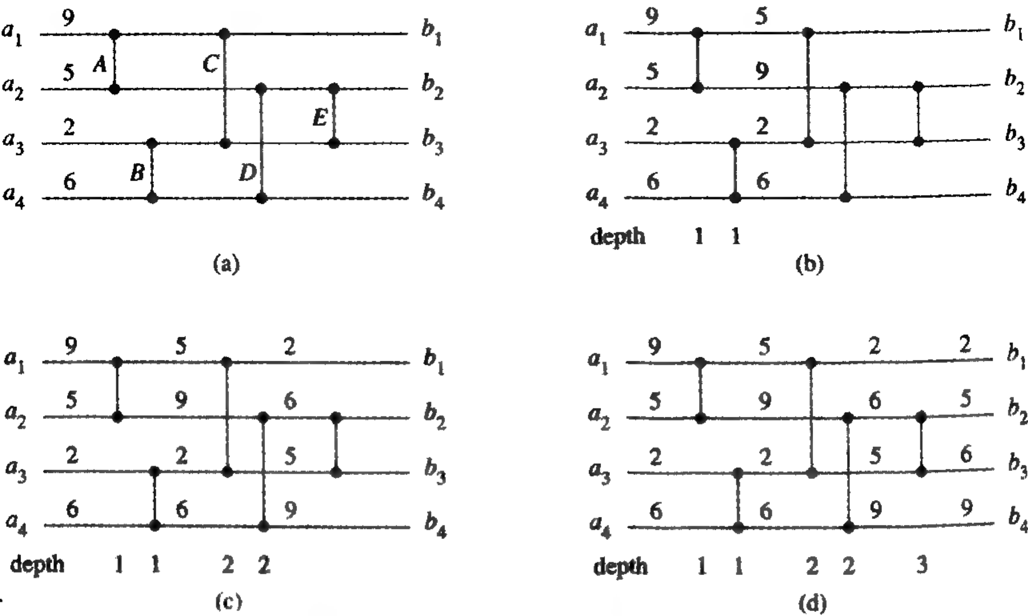
Ta mặc nhận mỗi bộ so sánh hoạt động trong $O(1)$ thời gian. Nói cách khác, ta mặc nhận thời gian giữa sự xuất hiện của các giá trị nhập x và y và việc sản xuất các giá trị kết xuất x' và y' là một hằng.

Một ***dây dẫn*** truyền một giá trị từ nơi này đến nơi khác. Các dây dẫn có thể nối kết xuất của một bộ so sánh với đầu vào của một bộ so sánh khác, mà bằng không chúng là các dây dẫn đầu vào mạng hoặc các dây dẫn kết xuất mạng. Suốt chương này, ta sẽ mặc nhận rằng một mạng so sánh chứa ***n dây dẫn đầu vào*** $a_1, a_2, \ldots, a_n$, qua đó các giá trị được sắp xếp sẽ nhập mạng, và ***n dây dẫn kết xuất*** $b_1, b_2, \ldots, b_n$, tạo ra các kết quả đã được mạng tính toán. Ngoài ra, ta sẽ nói về ***dãy đầu vào*** $\langle a_1, a_2, \ldots, a_n \rangle$ và ***dãy kết xuất*** $\langle b_1, b_2, \ldots, b_n \rangle$, ám chỉ các giá trị trên các dây dẫn đầu vào và kết xuất. Nghĩa là, ta dùng cùng tên cho cả dây dẫn lẫn giá trị mà nó mang tải. Chủ trương của chúng ta luôn rõ ràng đối với ngữ cảnh.

Hình 28.2 nêu một ***mạng so sánh***, là một tập hợp các bộ so sánh tương kết bằng các dây dẫn. Ta vẽ một mạng so sánh trên n đầu vào dưới dạng một tập hợp n các vạch ngang với các bộ so sánh được trải dài theo chiều dọc. Lưu ý, một vạch không biểu diễn một dây dẫn đơn lẻ, nhưng thay vì thế là một dãy các dây dẫn riêng biệt nối các bộ so sánh khác nhau. Ví dụ, vạch trên cùng trong Hình 28.2 biểu diễn ba dây

dẫn: dây dẫn đầu vào a_1, nối với một đầu vào của bộ so sánh *A*; một dây dẫn nối kết xuất đỉnh của bộ so sánh *A* với một đầu vào của bộ so sánh *C*; và dây dẫn kết xuất b_1, xuất phát từ kết xuất đỉnh của bộ so sánh *C*. Mỗi đầu vào của bộ so sánh được nối với một dây dẫn là một trong số *n* dây dẫn đầu vào $a_1, a_2,..., a_n$ của mạng hoặc được nối với kết xuất của một bộ so sánh khác. Cũng vậy, mỗi bộ so sánh kết xuất được nối với một dây dẫn là một trong số *n* dây dẫn kết xuất $b_1, b_2,..., b_n$ của mạng hoặc được nối với đầu vào của một bộ so sánh khác. Yêu cầu chính để tương kết các bộ so sánh đó là đồ thị của các tương kết phải là phi chu trình: nếu ta theo dõi một lộ trình từ kết xuất của một bộ so sánh đã cho đến đầu vào của một bộ so sánh khác đến kết xuất đến đầu vào, vân vân., lộ trình mà ta theo dõi không được quanh vòng lại trên chính nó và đi qua cùng bộ so sánh hai lần. Như vậy, như trong Hình 28.2, ta có thể vẽ một mạng so sánh với các đầu vào mạng bên trái và các kết xuất mạng bên phải; dữ liệu dời qua mạng từ trái qua phải.

Mỗi bộ so sánh chỉ tạo ra các giá trị kết xuất khi cả hai giá trị đầu vào



Hình 28.2 (a) Một mạng so sánh 4-nhập liệu, 4-kết xuất, mà thực tế là một mạng sắp xếp. Vào lúc 0, các giá trị nhập liệu được nêu sẽ xuất hiện trên bốn dây dẫn nhập liệu. **(b)** Vào lúc 1 các giá trị được nêu sẽ xuất hiện trên các kết xuất của các bộ so sánh *A* và *B*, tại độ sâu 1. **(c)** Vào lúc 2, các giá trị được nêu sẽ xuất hiện trên các kết xuất của các bộ so sánh *C* và *D*, tại độ sâu 2. Các dây dẫn kết xuất b_1 và b_4 giờ đây có các giá trị chung cuộc của chúng, nhưng các dây dẫn kết xuất b_2 và b_3 thì không. (d) Vào lúc 3, các giá trị được nêu sẽ xuất hiện trên các kết xuất của bộ so sánh *E*, tại độ sâu 3. Các dây dẫn kết xuất b_2 và b_3 giờ đây có các giá trị chung cuộc của chúng.

của nó đều sẵn có đối với nó. Ví dụ, trong Hình 28.2(a), giả sử dãy <9, 5, 2, 6> xuất hiện trên các dây dẫn đầu vào vào lúc 0. Như vậy, vào lúc 0, chỉ các bộ so sánh *A* và *B* mới có tất cả các giá trị đầu vào sẵn có của chúng. Giả sử mỗi bộ so sánh yêu cầu một đơn vị thời gian để tính toán các giá trị kết xuất của nó, các bộ so sánh *A* và *B* tạo ra các kết xuất của chúng vào lúc 1; các giá trị kết quả được nêu trong Hình 28.2(b). Lưu ý, các bộ so sánh *A* và *B* tạo ra các giá trị của chúng cùng một lúc, hoặc "song song." Giờ đây, vào lúc 1, các bộ so sánh *C* và *D*, chứ không phải *E*, có tất cả các giá trị đầu vào sẵn có của chúng. Một đơn vị thời gian sau đó, vào lúc 2, chúng tạo ra các kết xuất của chúng, như đã nêu trong Hình 28.2(c). Các bộ so sánh *C* và *D* cũng hoạt động song song. Kết xuất đỉnh của bộ so sánh *C* và kết xuất đáy của bộ so sánh *D* nối với các dây dẫn kết xuất b_1 và b_4 của mạng so sánh, theo thứ tự nêu trên, và do đó các dây dẫn kết xuất mạng này mang tải các giá trị chung cuộc của chúng vào lúc 2. Trong khi đó, vào lúc 2, bộ so sánh *E* có các đầu vào sẵn có của nó, và Hình 28.2(d) chứng tỏ nó tạo ra các giá trị kết xuất của nó vào lúc 3. Các giá trị này được mang tải trên các dây dẫn kết xuất mạng b_2 và b_3, và dãy kết xuất <2, 5, 6, 9> giờ đây hoàn thành.

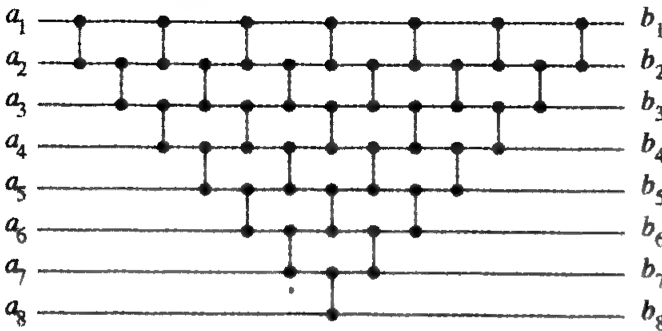
Dưới giả thiết cho rằng mỗi bộ so sánh chiếm thời gian đơn vị, ta có thể định nghĩa "thời gian thực hiện" của một mạng so sánh, nghĩa là, thời gian cần có để tất cả các dây dẫn kết xuất nhận được các giá trị của chúng một khi các dây dẫn đầu vào nhận được giá trị của chúng. Không chính thức, thời gian này là số lượng các bộ so sánh lớn nhất mà một thành phần nhập bất kỳ có thể đi qua khi nó di chuyển từ một dây dẫn đầu vào đến một dây dẫn kết xuất. Chính thức hơn, ta định nghĩa ***độ sâu*** của một dây dẫn như sau. Một dây dẫn đầu vào của một mạng so sánh có độ sâu 0. Giờ đây, nếu một bộ so sánh có hai dây dẫn đầu vào có các độ sâu d_x và d_y, thì các dây dẫn kết xuất của nó sẽ có độ sâu $\max(d_x, d_y) + 1$. Bởi ta không có chu trình nào của các bộ so sánh trong một mạng so sánh, nên độ sâu của một dây dẫn cũng được định nghĩa, và ta định nghĩa độ sâu của một bộ so sánh là độ sâu của các dây dẫn kết xuất của nó. Hình 28.2 nêu các độ sâu bộ so sánh. Độ sâu của một mạng so sánh là độ sâu cực đại của một dây dẫn kết xuất hoặc, một cách tương đương, là độ sâu cực đại của một bộ so sánh. Ví dụ, mạng so sánh của Hình 28.2 có độ sâu 3 bởi bộ so sánh *E* có độ sâu 3. Nếu mỗi bộ so sánh chiếm một đơn vị thời gian để tạo giá trị kết xuất của nó, và nếu các đầu vào mạng xuất hiện vào lúc 0, một bộ so sánh ở độ sâu d tạo ra các kết xuất của nó vào lúc *d*; do đó độ sâu của mạng bằng thời gian để mạng tạo ra các giá trị tại tất cả các dây dẫn kết xuất của nó.

Một ***mạng sắp xếp*** [sorting network] là một mạng so sánh mà dãy kết xuất tăng đơn điệu (nghĩa là, $b_1 \le b_2 \le \ldots \le b_n$) với mọi dãy đầu vào.

Tất nhiên, không phải mọi mạng so sánh đều là mạng sắp xếp, nhưng mạng của Hình 28.2 thì tất đúng. Để biết tại sao, ta nhận thấy sau thời gian 1, cực tiểu của bốn giá trị đầu vào đã được tạo bởi kết xuất đỉnh của bộ so sánh *A* hoặc kết xuất đỉnh của bộ so sánh *B*. Do đó, sau thời gian 2, nó phải nằm kết xuất đỉnh của bộ so sánh *C*. Một đối số đối xứng chứng tỏ sau thời gian 2, cực đại của bốn giá trị đầu vào đã được tạo bởi kết xuất đáy của bộ so sánh *D*. Phần còn lại của bộ so sánh *E* đó là bảo đảm hai giá trị giữa choán các vị trí kết xuất đúng đắn của chúng, sẽ xảy ra vào lúc 3.

Một mạng so sánh cũng giống như một thủ tục ở chỗ nó chỉ định các phép so sánh phải xảy ra như thế nào, nhưng khác với một thủ tục ở chỗ kích cỡ vật lý củanó tùy thuộc vào số lượng đầu vào và kết xuất. Do đó, ta sẽ thực tế mô tả các "họ" mạng so sánh. Ví dụ, mục tiêu của chương này đó là phát triển một họ SORTER gồm các mạng sắp xếp hiệu quả . Ta chỉ định một mạng đã cho trong một họ theo tên họ và số lượng đầu vào (bằng với số lượng kết xuất). Ví dụ, mạng sắp xếp *n*-đầu vào, *n*-kết xuất trong họ SORTER có tên là SORTER[*n*].

Bài tập



Hình 28.3 Một mạng sắp xếp dựa trên sắp xếp chèn để dùng trong Bài tập 28.1-6.

28.1-1

Nêu các giá trị xuất hiện trên tất cả các dây dẫn của mạng của Hình 28.2 khi nó có dãy đầu vào <9, 6, 5, 2>.

28.1-2

Cho *n* là một lũy thừa chính xác của 2. Nêu cách kiến tạo một mạng so sánh *n*-đầu vào, *n*-kết xuất có độ sâu lg *n* ở đó dây dẫn kết xuất đỉnh luôn mang tải giá trị đầu vào cực tiểu và dây dẫn kết xuất đáy luôn mang tải giá trị đầu vào cực đại.

28.1-3

Giáo sư Nielsen cho rằng nếu ta bổ sung một bộ so sánh vào bất kỳ đâu trong một mạng sắp xếp, mạng kết quả cũng sẽ sắp xếp. Chứng tỏ giáo sư đã lầm khi bổ sung một bộ so sánh vào mạng của Hình 28.2 theo cách mà mạng kết quả không sắp xếp mọi phép hoán vị đầu vào.

28.1-4

Chứng minh bất kỳ mạng sắp xếp nào on n đầu vào đều có độ sâu ít nhất là lg n.

28.1-5

Chứng minh số lượng các bộ so sánh trong một mạng sắp xếp bất kỳ ít nhất là $\Omega(n \lg n)$.

28.1-6

Xét mạng so sánh nêu trong Hình 28.3. Chứng minh nó thực tế là một mạng sắp xếp, và mô tả cấu trúc của nó liên quan như thế nào với cấu trúc của sắp xếp chèn (Đoạn 1.1).

28.1-7

Ta có thể biểu diễn một mạng so sánh n-đầu vào có c bộ so sánh dưới dạng một danh sách gồm c cặp số nguyên trong miền từ 1 đến n. Nếu hai cặp chứa một số nguyên chung, thứ tự của các bộ so sánh tương ứng trong mạng được xác định bởi thứ tự của các cặp trong danh sách. Căn cứ vào phép biểu diễn này, mô tả một thuật toán $O(n+c)$ thời gian (nối tiếp) để xác định độ sâu của một mạng so sánh.

*28.1-8 **

Giả sử ngoài kiểu chuẩn của bộ so sánh, ta giới thiệu một bộ so sánh lộn ngược "upside-down" tạo ra kết xuất cực tiểu cho việc đúng đắn trên các con số nhập tùy ý.

28.2 Nguyên lý Zero - một

Nguyên lý Zero - một (Zero - one principle) nói rằng nếu một mạng sắp xếp làm việc đúng đắn khi mỗi đầu vào được rút từ tập hợp {0,1}, thì nó làm việc đúng đắn trên các con số nhập tùy ý. (Các con số có thể là số nguyên, số thực, hoặc, nói chung, bất kỳ tập hợp giá trị nào từ một tập hợp sắp xếp theo tuyến tính bất kỳ.) Khi ta kiến tạo các mạng sắp xếp và các mạng so sánh khác, nguyên lý zero-một sẽ cho phép ta tập trung vào phép toán của chúng với các dãy đầu vào chỉ toàn các 0 và 1.

Một khi đã kiến tạo một mạng sắp xếp và chứng minh nó có thể sắp xếp tất cả các dãy zero-một, ta sẽ dựa vào nguyên lý zero-một để chứng tỏ nó sắp xếp đúng đắn các dãy giá trị tùy ý.

Phần chứng minh của nguyên lý zero-một dựa vào khái niệm của một hàm tăng đơn điệu (Đoạn 2.2).

Bổ đề 28.1

Nếu một mạng so sánh biến đổi dãy đầu vào $a = \langle a_1, a_2, \ldots, a_n \rangle$ thành dãy kết xuất $b = (b_1, b_2, \ldots, b_n)$, thì với một hàm tăng đơn điệu f, mạng biến đổi dãy đầu vào $f(a) = \langle f(a_1), f(a_2), \ldots, f(a_n \rangle)$ thành dãy kết xuất $f(b) = (f(b_1), f(b_2), \ldots, f(b_n) \rangle$.

Chứng minh Trước tiên ta sẽ chứng minh biện luận cho rằng nếu f là một hàm tăng đơn điệu, thì một bộ so sánh đơn có các đầu vào $f(x)$ và $f(y)$ sẽ tạo ra các kết xuất $f(\min(x, y))$ và $f(\max(x, y))$. Sau đó, ta dùng phương pháp quy nạp để chứng minh bổ đề.

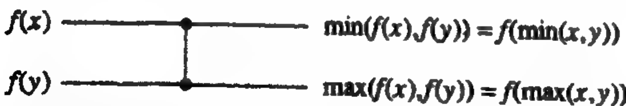
Để chứng minh biện luận, hãy xét một bộ so sánh có các giá trị đầu vào là x và y. Kết xuất trên của bộ so sánh là $\min(x, y)$ và kết xuất dưới là $\max(x, y)$. Giả sử giờ đây ta áp dụng $f(x)$ và $f(y)$ cho các đầu vào của bộ so sánh, như đã nêu trong Hình 28.4. Phép toán của bộ so sánh cho ra giá trị $\min(f(x), f(y))$ trên kết xuất trên và giá trị $\max(f(x), f(y))$ trên kết xuất dưới. Bởi f tăng đơn điệu, nên $x \le y$ hàm ý $f(x) \le f(y)$. Kết quả là, ta có các đồng nhất thức

$$\min(f(x), f(y)) = f(\min(x,y)) ,$$

$$\max(f(x), f(y)) = f(\max(x,y)) .$$

Như vậy, bộ so sánh tạo ra các giá trị $f(\min(x,y))$ và $f(\max(x,y))$ khi $f(x)$ và $f(y)$ là đầu vào của nó, hoàn tất phần chứng minh của biện luận.

Ta có thể dùng phương pháp quy nạp trên độ sâu của mỗi dây dẫn trong một mạng so sánh chung để chứng minh một kết quả mạnh hơn so với phát biểu của bổ đề: nếu một dây dẫn mặc nhận giá trị a_i khi dãy đầu vào a được áp dụng cho mạng, thì nó mặc nhận giá trị $f(a_i)$ khi dãy đầu vào $f(a)$ được áp dụng. Bởi các dây dẫn kết xuất được gộp trong phát biểu này, nên việc chứng minh nó sẽ chứng minh bổ đề.



Hình 28.4 Phép toán của bộ so sánh trong phần chứng minh của Bổ đề 28.1. Hàm f tăng đơn điệu.

Về cơ bản, ta xét một dây dẫn tại độ sâu 0, nghĩa là, một dây dẫn đầu vào a_i. Kết quả không có gì đáng nói: khi $f(a)$ được áp dụng cho mạng, dây dẫn đầu vào mang tải $f(a_i)$. Với bước quy nạp, ta xét một dây dẫn tại độ sâu d, ở đó $d \geq 1$. Dây dẫn là kết xuất của một bộ so sánh tại độ sâu d, và các dây dẫn đầu vào cho bộ so sánh này nằm tại một độ sâu hoàn toàn nhỏ hơn d. Do đó, theo giả thuyết quy nạp, nếu các dây dẫn đầu vào cho bộ so sánh mang tải các giá trị a_i và a_j khi dãy đầu vào a được áp dụng, thì chúng mang tải $f(a_i)$ và $f(a_j)$ khi dãy đầu vào $f(a)$ được áp dụng. Theo biện luận trên đây của chúng ta, các dây dẫn kết xuất của bộ so sánh này sẽ mang tải $f(\min(a_i, a_j))$ và $f(\max(a_i, a_j))$. Bởi chúng mang tải $\min(a_i, a_j)$ và $\max(a_i, a_j)$ khi dãy đầu vào là a, bổ đề được chứng minh.

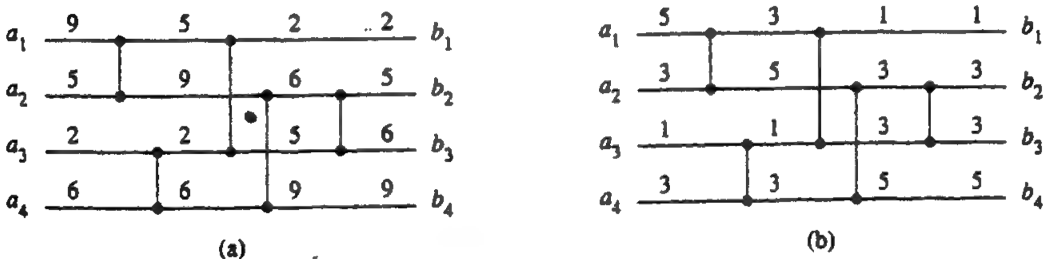
Để lấy ví dụ về ứng dụng của Bổ đề 28.1, Hình 28.5 nêu mạng sắp xếp từ Hình 28.2 với hàm tăng đơn điệu $f(x) = \lceil x/2 \rceil$ được áp dụng cho các đầu vào. Giá trị trên mọi dây dẫn là f được áp dụng cho giá trị trên cùng dây dẫn trong Hình 28.2.

Khi một mạng so sánh là một mạng sắp xếp, Bổ đề 28.1 oho phép ta chứng minh kết quả đáng lưu ý dưới đây.

Định lý 28.2 (Nguyên lý zero-một)

Nếu một mạng so sánh có n đầu vào sắp xếp đúng đắn tất cả 2^n dãy khả dĩ các 0 và 1, thì nó sắp xếp đúng đắn tất cả các dãy con số tùy ý.

Chứng minh Vì sự mâu thuẫn, ta giả sử mạng sắp xếp tất cả các dãy zero-một, nhưng ở đó tồn tại một dãy các con số tùy ý mà mạng sắp xếp không đúng đắn. Nghĩa là, ở đó tồn tại một dãy đầu vào $\langle a_1, a_2, \ldots, a_n \rangle$ chứa các thành phần a_i và a_j sao cho $a_i < a_j$, nhưng mạng đặt a_j trước



Hình 28.5 (a) Mạng sắp xếp từ Hình 28.2 với dãy nhập liệu <9,5,2,6>. **(b)** Cùng mạng sắp xếp với hàm tăng đơn điệu $f(x) = f(\lceil x/2 \rceil)$ được áp dụng cho các nhập liệu. Mỗi dây dẫn trong mạng này có giá trị của f được áp dụng cho giá trị trên dây dẫn tương ứng trong (a).

a_i trong dãy kết xuất. Ta định nghĩa một hàm tăng đơn điệu f là

$$f(x) = \begin{cases} 0 & \text{nếu } x \leq a_i, \\ 1 & \text{nếu } x > a_i. \end{cases}$$

Do mạng đặt a_j trước a_i trong dãy kết xuất khi $<a_1, a_2,..., a_n>$ là đầu vào, nên theo Bổ đề 28.1 nó đặt $f(a_j)$ trước $f(a_i)$ trong dãy kết xuất khi $(f(a_1), f(a_2),..., f(a_n))$ là đầu vào. Nhưng do $<f(a_j) = 1$ và $f(a_i)> = 0$, nên ta có sự mâu thuẫn cho rằng mạng không thể sắp xếp đúng đắn dãy zero-một $<f(a_1), f(a_2),..., f(a_n)>$.

Bài tập

28.2-1

Chứng minh việc áp dụng một hàm tăng đơn điệu cho một dãy sắp xếp sẽ tạo ra một dãy sắp xếp.

28.2-2

Chứng minh một mạng so sánh có n đầu vào sẽ sắp xếp đúng đắn dãy đầu vào $<n, n-1,..., 1>$ nếu và chỉ nếu nó sắp xếp đúng đắn $n - 1$ dãy zero-một <1,0,0,...,0,0>, <1, 1,0,...,0,0>,..., <1, 1, 1,..., 1,0>.

28.2-3

Dùng nguyên lý zero-một để chứng minh mạng so sánh nêu trong Hình 28.6 là một mạng sắp xếp.

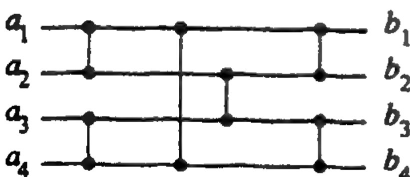
28.2-4

Phát biểu và chứng minh một dạng tương tự của nguyên lý zero-một cho một mô hình cây-quyết định. (*Mách nước:* Bảo đảm điều quản đẳng thức đúng đắn.)

28.2-5

Chứng minh một mạng sắp xếp n-đầu vào phải chứa ít nhất một bộ so sánh giữa các dòng thứ i và thứ $(i + 1)$ với tất cả $i = 1, 2,..., n - 1$.

Bước đầu tiên trong tiến trình kiến tạo một mạng sắp xếp hiệu quả



Hình 28.6 Một mạng sắp xếp để sắp xếp 4 con số.

28.3 Mạng sắp xếp bitonic

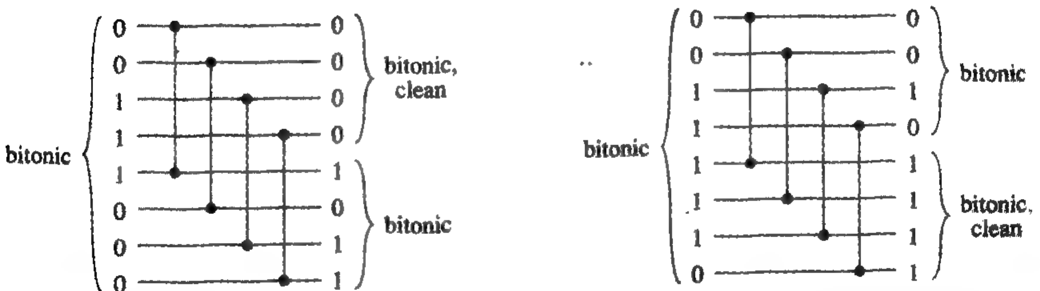
đó là kiến tạo một mạng so sánh có thể sắp xếp bất kỳ ***dãy bitonic***: một dãy tăng đơn điệu rồi giảm đơn điệu, nếu giảm đơn điệu rồi tăng đơn điệu. Ví dụ, <1, 4, 6, 8, 3, 2> và <9, 8, 3, 2, 4, 6> là hai dãy bitonic. Các dãy zero-một là bitonic có một cấu trúc đơn giản. Chúng có dạng $0^i1^j0^k$ hoặc dạng $1^i0^j1^k$, với một $i, j, k \geq 0$. Lưu ý, một dãy tăng đơn điệu hoặc giảm đơn điệu cũng là một dãy bitonic.

Bộ sắp xếp bitonic mà ta sẽ kiến tạo là một mạng so sánh sắp xếp các dãy bitonic gồm các 0 và 1. Bài tập 28.3-6 yêu cầu bạn chứng tỏ bộ sắp xếp bitonic có thể sắp xếp các dãy bitonic các con số tùy ý.

Bộ bán xóa

Một bộ sắp xếp bitonic bao gồm vài giai đoạn, mỗi giai đoạn được gọi là một ***bộ nửa sạch*** [half-cleaner]. Mỗi bộ nửa sạch là một mạng so sánh có độ sâu 1 ở đó dòng đầu vào i được so sánh với dòng $i + n/2$ với $i = 1, 2, \ldots, n/2$. (Ta mặc nhận rằng n là chẵn.) Hình 28.7 nêu HALF-CLEANER[8], bộ nửa sạch với 8 đầu vào và 8 kết xuất.

Khi một dãy bitonic các 0 và 1 được áp dụng làm đầu vào cho một bộ nửa sạch, bộ nửa sạch tạo ra một dãy kết xuất ở đó các giá trị nhỏ hơn nằm trong nửa trên, các giá trị lớn hơn nằm trong nửa dưới, và cả hai nửa đều là bitonic. Thực vậy, ít nhất một trong hai nửa là sạch—bao gồm tất cả các 0 hoặc tất cả các 1—và chính từ tính chất này ta suy ra tên "bộ nửa sạch." (Lưu ý, tất cả các dãy sạch đều là bitonic.) Bổ đề dưới đây chứng minh các tính chất này của các bộ nửa sạch.



Hình 28.7 Mạng so sánh HALF-CLEANER[8]. Có nêu hai giá trị nhập liệu và kết xuất zero-một làm mẫu khác nhau. Nhập liệu được mặc nhận là bitonic. Một bộ nửa sạch bảo đảm mọi thành phần kết xuất của nửa trên ít nhất nhỏ bằng mọi thành phần kết xuất của nửa dưới. Hơn nữa, cả hai nửa là bitonic, và ít nhất một nửa là sạch.

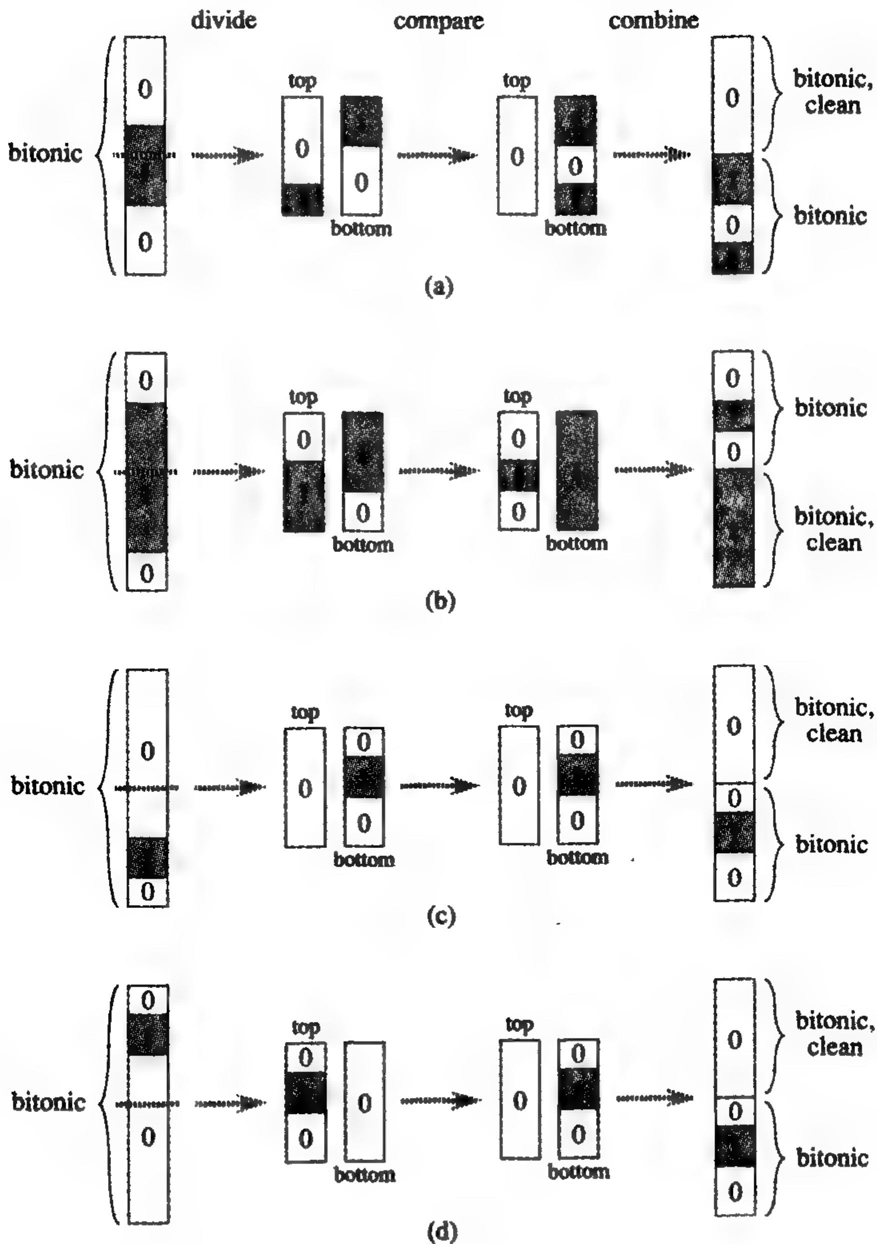
Bổ đề 28.3

Nếu đầu vào cho một bộ nửa sạch là một dãy bitonic gồm các 0 và 1, thì kết xuất thỏa các tính chất sau đây: cả nửa trên lẫn nửa dưới đều là bitonic, mọi thành phần trong nửa trên ít nhất nhỏ bằng mọi thành phần của nửa dưới, và ít nhất một nửa là sạch.

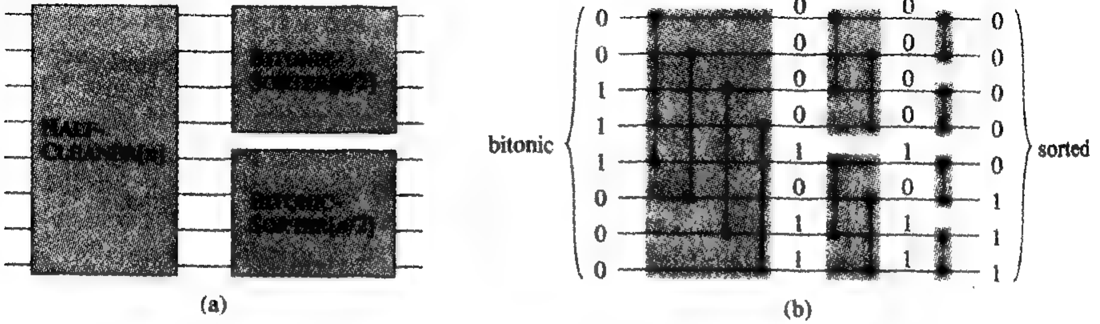
Chứng minh Mạng so sánh HALF-CLEANER[n] so sánh các đầu vào i và $i + n/2$ với $i = 1, 2,..., n/2$. Không để mất tính tổng quát, giả sử rằng đầu vào có dạng 00...011...100...0. (Tình huống ở đó đầu vào có dạng 11...100...011...1 là đối xứng.) Có ba trường hợp khả dĩ tùy thuộc vào khối các 0 hoặc 1 liên tiếp nơi trung điểm $n/2$ xảy ra, và một trong các trường hợp này (ở đó trung điểm xảy ra trong khối các 1) được tách thêm thành hai trường hợp. Bốn trường hợp được nêu trong Hình 28.8. Trong mỗi trường hợp đã nêu, bổ đề đều đứng vững.

Bộ sắp xếp bitonic

Nhờ tổ hợp một cách đệ quy các bộ nửa sạch, như đã nêu trong Hình 28.9, ta có thể xây dựng một ***bộ sắp xếp bitonic***, là một mạng sắp xếp các dãy bitonic. Giai đoạn đầu của BITONIC-SORTER[n] bao gồm HALF-CLEANER[n], mà, theo Bổ đề 28.3, tạo ra hai dãy bitonic có nửa kích cỡ sao cho mọi thành phần trong nửa trên ít nhất nhỏ bằng mọi thành phần trong nửa dưới. Như vậy, ta có thể hoàn thành đợt sắp xếp bằng cách dùng hai bản sao của BITONIC-SORTER[$n/2$] để sắp xếp hai nửa một cách đệ quy. Trong Hình 28.9(a), phép đệ quy đã được nêu rõ rệt, và trong Hình 28.9(b), phép đệ quy đã được trải rộng để nêu các bộ nửa sạch nhỏ hơn dần tạo thành phần còn lại của bộ sắp xếp bitonic. Độ sâu $D(n)$ của BITONIC-SORTER[n] được căn cứ vào phép truy toán



Hình 28.8 Các phép so sánh khả dĩ trong HALF-CLEANER[n]. Dãy nhập liệu được mặc nhận là một dãy bitonic gồm các 0 và 1, và không để mất tính tổng quát, ta mặc nhận rằng nó có dạng 00...011...100...0. Các dãy con các 0 được tô trắng, và các dãy con các 1 được tô xám. Ta có thể xem n nhập liệu như đang được chia thành hai nửa sao cho với $i = 1, 2, ..., n/2$, các nhập liệu i và $i + n/2$ được so sánh. **(a)-(*b*)** Các trường hợp ở đó xảy ra phép chia trong dãy con giữa của các 1. **(c)-(d)** Các trường hợp ở đó xảy ra phép chia trong một dãy con các 0. Với tất cả các trường hợp, mọi thành phần trong nửa trên ít nhất nhỏ bằng mọi thành phần trong nửa dưới, cả hai nửa đều là bitonic, và ít nhất một nửa là sạch.



Hình 28.9 Mạng so sánh BITONIC-SORTER[n], nêu ở đây với n = 8. **(a)** Phần kiến tạo đệ quy: HALF-CLEANER[n] theo sau là hai bản sao của BITONIC-SORTER[n/2] hoạt động song song. **(b)** Mạng sau khi trải rộng phép đệ quy. Mỗi bộ nửa sạch được tô bóng. Các giá trị zero-một mẫu được nêu trên các dây dẫn.

$$D(n) = \begin{cases} 0 & \text{nếu } n=1, \\ D(n/2) + 1 & \text{nếu } n = 2^k \text{ và } k \geq 1 \end{cases}$$

mà nghiệm của nó là $D(n) = \lg n$.

Như vậy, một dãy bitonic zero-một có thể được BITONIC-SORTER sắp xếp, có một độ sâu là lg n. Nó dẫn đến tính tương tự của nguyên lý zero-một được cho làm Bài tập 28.3-6 rằng mọi dãy bitonic các con số tùy ý đều có thể được sắp xếp bằng mạng này.

Bài tập

28.3-1

Có bao nhiêu các dãy bitonic các 0 và 1?

28.3-2

Chứng tỏ BITONIC-SORTER[n], ở đó n là một lũy thừa chính xác của 2, chứa $\Theta(n \lg n)$ bộ so sánh.

28.3-3

Mô tả cách kiến tạo một bộ sắp xếp bitonic O(lgn) độ sâu khi con số n đầu vào không phải là một lũy thừa chính xác của 2.

28.3-4

Nếu đầu vào cho một bộ nửa sạch là một dãy bitonic các con số tùy ý, hãy chứng minh kết xuất thỏa các tính chất sau đây: cả nửa trên lẫn nửa dưới đều là bitonic, và mọi thành phần trong nửa trên ít nhất nhỏ bằng mọi thành phần trong nửa dưới.

28.3-5

Xét hai dãy các 0 và 1. Chứng minh nếu mọi thành phần trong một dãy ít nhất nhỏ bằng mọi thành phần trong dãy kia, thì một trong hai dãy là sạch.

28.3-6

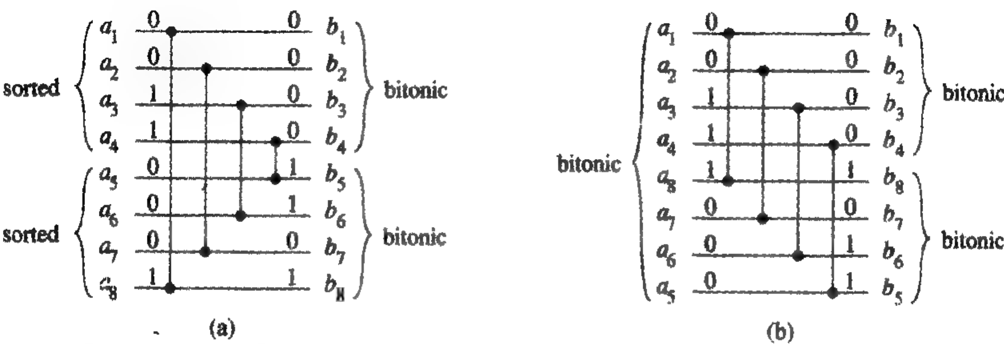
Chứng minh dạng tương tự dưới đây của nguyên lý zero-một đối với các mạng sắp xếp bitonic: một mạng so sánh có thể sắp xếp bất kỳ dãy bitonic các 0 và 1 nào cũng có thể sắp xếp một dãy bitonic bất kỳ gồm các con số tùy ý.

28.4 Một mạng trộn

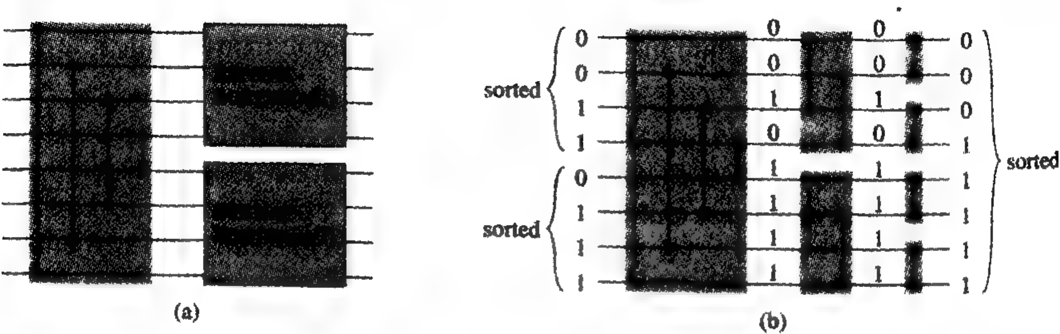
Mạng sắp xếp của chúng ta sẽ được kiến tạo từ các ***mạng trộn*** [merging network], là các mạng có thể trộn hai dãy đầu vào sắp xếp thành một dãy kết xuất sắp xếp. Ta sửa đổi BITONIC-SORTER[n] để tạo mạng trộn MERGER[n]. Giống như bộ sắp xếp bitonic, ta chỉ chứng minh tính đúng đắn của mạng trộn cho các đầu vào là các dãy zero-một. Bài tập 28.4-1 yêu cầu bạn nêu cách mở rộng phần chứng minh sang các giá trị đầu vào tùy ý.

Mạng trộn dựa trên trực giác sau đây. Cho hai dãy sắp xếp, nếu ta đảo thứ tự của dãy thứ hai rồi ghép nối hai dãy, dãy kết quả sẽ là bitonic. Ví dụ, cho các dãy zero-một đã sắp xếp $X = 00000111$ và $Y = 00001111$, ta đảo Y để có $Y^R = 11110000$. Ghép nối X và Y^R sẽ cho ra 0000011111110000, là bitonic. Như vậy, để trộn hai đầu vào các dãy X và Y, ta chỉ cần thực hiện một đợt sắp xếp bitonic trên X được ghép nối với Y^R.

Ta có thể kiến tạo MERGER[n] bằng cách sửa đổi bộ nửa sạch đầu tiên của BITONIC-SORTER[n]. Điểm chính đó là thực hiện phép đảo hoàn toàn trên nửa thứ hai của các đầu vào. Cho hai dãy sắp xếp $<a_1, a_2, \ldots, a_{n/2}>$ và $<a_{n/2+1}, a_{n/2+2}, a_n>$ để trộn, ta muốn hiệu ứng sắp xếp bitonic trên dãy $<a_1, a_2, \ldots, a_{n/2}, a_n, a_{n-1}, \ldots, a_{n/2+1}>$. Bởi bộ nửa sạch của BITONIC-SORTER[n] so sánh các đầu vào i và $n/2 + i$, với $i = 1, 2, \ldots, n/2$, ta thực hiện giai đoạn đầu của mạng trộn so sánh các đầu vào i và $n - i + 1$. Hình 28.10 nêu sự tương ứng. Điểm tinh tế duy nhất đó là thứ tự của các kết xuất từ đáy của giai đoạn đầu của MERGER[n] được đảo ngược so sánh với thứ tự của các kết xuất từ một bộ nửa sạch bình thường. Tuy nhiên, do phép đảo của một dãy bitonic là bitonic, nên các kết xuất đỉnh và đáy của giai đoạn đầu của mạng trộn thỏa các tính chất trong Bổ đề 28.3, và như vậy đỉnh và đáy có thể được sắp xếp bitonic song song để tạo ra kết xuất sắp xếp của mạng trộn.



Hình 28.10 So sánh giai đoạn đầu của MERGER[n] và HALF-CLEANER[n], với $n = 8$. **(a)** Giai đoạn đầu của MERGER[n] biến đổi hai dãy nhập liệu monotonic $<a_1, a_2,..., a_{n/2}>$ và $<a_{n/2+1}, a_{n/2+2}, ..., a_n>$ thành hai dãy bitonic $<b_1, b_2,..., b_{n/2}>$ và $<b_{n/2+1}, b_{n/2+2}, ..., b_n>$. **(b)** Phép toán tương đương cho HALF-CLEANER[n]. Dãy nhập liệu bitonic $<a_1, a_2,..., a_{n/2-1}, a_{n/2}, a_n, a_{n-1}, ..., a_{n/2+2}, a_{n/2+1}>$ được biến đổi thành hai dãy bitonic $<b_1, b_2,..., b_{n/2}>$ và $<b_n, b_{n-1}, ..., b_{n/2+1}>$.



Hình 28.11 Một mạng trộn hai dãy nhập liệu sắp xếp thành một dãy sắp xếp kết xuất. Mạng MERGER[n] có thể được xem như BITONIC-SORTER[n] với bộ nửa sạch đầu tiên được thay đổi để so sánh các nhập liệu i và $n - i + 1$ với $i = 1, 2,..., n/2$. Ở đây, $n = 8$. **(a)** Mạng được phân tích thành giai đoạn đầu theo sau là hai bản sao song song của BITONIC-SORTER[$n/2$]. **(b)** Cùng mạng với phép đệ quy đã trải rộng. Các giá trị zero-một mẫu được nêu trên các dây dẫn, và các giai đoạn được tô bóng.

Mạng trộn kết quả được nêu trong Hình 28.11. Chỉ giai đoạn đầu của MERGER[n] là khác với BITONIC-SORTER[n]. Bởi vậy, độ sâu của MERGER[n] là lg n, giống như của BITONIC-SORTER[n].

Bài tập

28.4-1

Chứng minh một dạng tương tự của nguyên lý zero-một cho các mạng trộn. Cụ thể, chứng tỏ một mạng so sánh có thể trộn hai dãy tăng đơn điệu bất kỳ gồm các 0 và 1 có thể trộn hai dãy tăng đơn điệu bất kỳ gồm các con số tùy ý.

28.4-2

Phải áp dụng bao nhiêu dãy đầu vào zero-một khác nhau cho đầu vào của một mạng so sánh để xác minh nó là một mạng trộn?

28.4-3

Chứng tỏ một mạng bất kỳ có thể trộn 1 mục với n - 1 mục để tạo ra một dãy sắp xếp có chiều dài n phải có độ sâu ít nhất là lg n.

28.4-4 *

Xét một mạng trộn có các đầu vào $a_1, a_2, ..., a_n$, với n một lũy thừa chính xác của 2, ở đó hai dãy monotonic được trộn là $<a_1, a_3, ..., a_{n-1}>$ và $<a_2, a_4, ..., a_n>$. Chứng minh số lượng các bộ so sánh trong kiểu mạng trộn này là $\Omega(n \lg n)$. Tại sao đây là một cận dưới đáng quan tâm? (*Mách nước:* Phân hoạch các bộ so sánh thành ba tập hợp.)

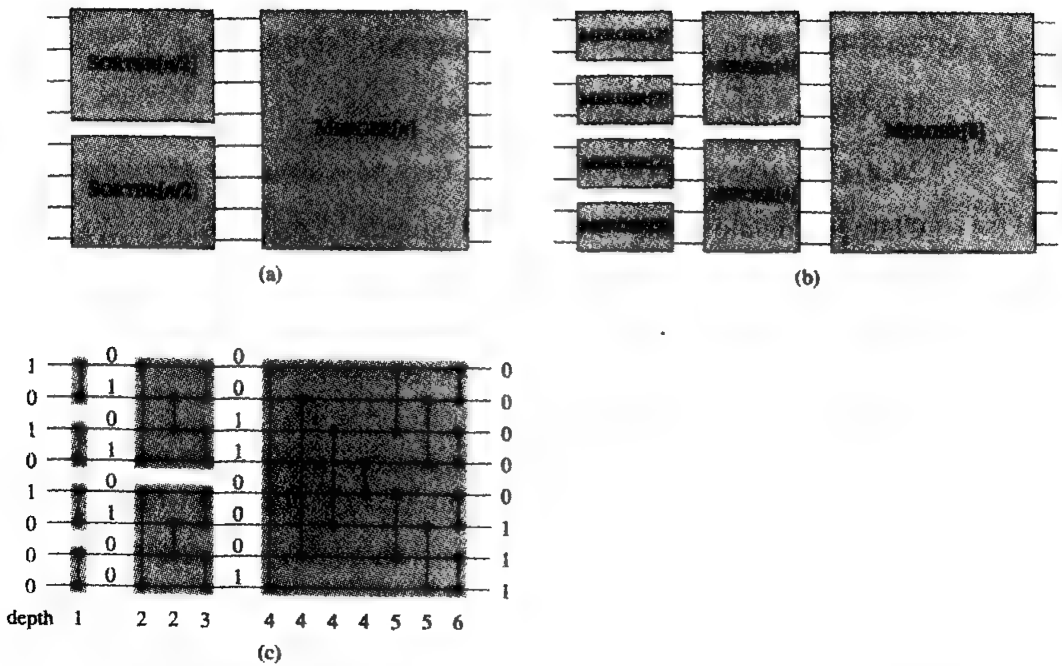
28.4-5 *

Chứng minh mọi mạng trộn, bất chấp thứ tự của các đầu vào, đều yêu cầu $\Omega(n \lg n)$ bộ so sánh.

28.5 Mạng sắp xếp

Giờ đây ta có tất cả các công cụ cần thiết để kiến tạo một mạng có thể sắp xếp bất kỳ dãy đầu vào nào. Mạng sắp xếp SORTER[n] sử dụng mạng trộn để thực thi một phiên bản song song của kiểu sắp xếp trộn trong Đoạn 1.3.1. Phần kiến tạo và phép toán của mạng sắp xếp được minh họa trong Hình 28.12.

Hình 28.12(a) nêu phần kiến tạo đệ quy của SORTER[n]. n thành phần nhập được sắp xếp bằng cách dùng hai bản sao của SORTER [$n/2$] bằng các mạng trộn thực tế.



Hình 28.12 Mạng sắp xếp SORTER[n] được kiến tạo bằng cách tổ hợp đệ quy các mạng trộn. **(a)** Phần kiến tạo đệ quy. **(b)** Trải rộng phép đệ quy. **(c)** Thay các hộp MERGER bằng các mạng trộn thực tế. Độ sâu của mỗi bộ so sánh được nêu ra, và các giá trị zero-một mẫu được nêu trên các dây dẫn.

Dữ liệu đi qua lg n giai đoạn trong mạng SORTER[n]. Từng đầu vào riêng lẻ cho mạng đã là một dãy 1-thành phần đã sắp xếp. Giai đoạn đầu của SORTER[n] bao gồm $n/2$ bản sao của MERGER[2] làm việc song song để trộn các cặp dãy 1-thành phần để tạo các dãy sắp xếp có chiều dài 2. Giai đoạn thứ hai bao gồm $n/4$ bản sao của MERGER[4] trộn các cặp các dãy 2-thành phần đã sắp xếp này để tạo các dãy sắp xếp có chiều dài 4. Nói chung, với $k = 1, 2, ..., \lg n$, giai đoạn k bao gồm $n/2^k$ bản sao của MERGER[2^k] trộn các cặp các dãy 2^{k-1} -thành phần đã sắp xếp để tạo các dãy sắp xếp có chiều dài 2^k. Vào giai đoạn cuối, một dãy sắp xếp bao gồm tất cả các giá trị đầu vào được tạo. Mạng sắp xếp này có thể được nêu bằng phương pháp quy nạp để sắp xếp các dãy zero-một, và bởi vậy, theo nguyên lý zero-một (Định lý 28.2), nó có thể sắp xếp các giá trị tùy ý.

Ta có thể phân tích độ sâu của mạng sắp xếp một cách đệ quy. Độ sâu $D(n)$ của SORTER[n] là độ sâu $D(n/2)$ của SORTER[$n/2$] (có hai bản sao SORTER[$n/2$], nhưng chúng hoạt động song song) cộng với độ sâu lg n của MERGER[n]. Bởi vậy, độ sâu của SORTER[n] được căn cứ vào phép truy toán

$$D(n) = \begin{cases} 0 & \text{nếu } n = 1 , \\ D(n/2) + \lg n & \text{nếu } n = 2^k \text{ và } k \geq 1 , \end{cases}$$

mà giải pháp của nó là $D(n) = \Theta(\lg^2 n)$. Như vậy, ta có thể sắp xếp n con số song song trong $O(\lg^2 n)$ thời gian.

Bài tập

28.5-1

Có bao nhiêu bộ so sánh trong *SORTER*[n]?

28.5-2

Chứng tỏ độ sâu của *SORTER*[n] chính xác là $(\lg n)(\lg n + 1)/2$.

28.5-3

Giả sử ta sửa đổi một bộ so sánh để nhận hai danh sách đã sắp xếp có chiều dài k làm các đầu vào, trộn chúng, và kết xuất k lớn nhất sang kết xuất "max" của nó và k nhỏ nhất sang kết xuất "min" của nó. Chứng tỏ mọi mạng sắp xếp trên n đầu vào có các bộ so sánh đã sửa đổi theo cách này có thể sắp xếp nk con số, mặc nhận mỗi đầu vào cho mạng là một danh sách đã sắp xếp có chiều dài k.

28.5-4

Giả sử ta có $2n$ thành phần $\langle a_1, a_2, \ldots, a_{2n} \rangle$ và muốn phân hoạch chúng thành n nhỏ nhất và n lớn nhất. Chứng minh ta có thể thực hiện điều này trong độ sâu bổ sung bất biến sau khi sắp xếp tách biệt $\langle a_1, a_2, \ldots, a_n \rangle$ và $\langle a_{n+1}, a_{n+2}, \ldots, a_{2n} \rangle$.

28.5-5 *

Cho $S(k)$ là độ sâu của một mạng sắp xếp có k đầu vào, và cho $M(k)$ là độ sâu của một mạng trộn có $2k$ đầu vào. Giả sử ta có một dãy n con số để sắp xếp và biết rằng mọi số nằm tại k vị trí của vị trí đúng đắn củanó trong thứ tự sắp xếp. Chứng tỏ ta có thể sắp xếp n con số trong độ sâu $S(k) + 2M(k)$.

28.5-6 *

Ta có thể sắp xếp các thành phần của một ma trận $m \times m$ bằng cách lập lại thủ tục dưới đây k lần:

1. Sắp xếp mỗi hàng đánh số lẻ theo thứ tự tăng đơn điệu.
2. Sắp xếp mỗi hàng đánh số chẵn theo thứ tự giảm đơn điệu.
3. Sắp xếp mỗi cột theo thứ tự tăng đơn điệu.

Ta cần bao nhiêu lần lặp lại k để thủ tục này sắp xếp, và đâu là khuôn mẫu của kết xuất đã sắp xếp?

Các Bài Toán

28-1 Các mạng sắp xếp chuyển vị

Một mạng so sánh là một ***mạng chuyển vị*** nếu mỗi bộ so sánh với các dòng kề, như trong mạng của Hình 28.3.

a. Chứng tỏ mọi mạng chuyển vị có n đầu vào sắp xếp sẽ có $\Omega(n^2)$ bộ so sánh.

b. Chứng minh một mạng chuyển vị có n đầu vào là một mạng sắp xếp nếu và chỉ nếu nó sắp xếp dãy $\langle n, n - 1, ..., 1\rangle$. (*Mách nước*: Dùng một lập luận quy nạp tương tự như trong phần chứng minh của Bổ đề 28.1.)

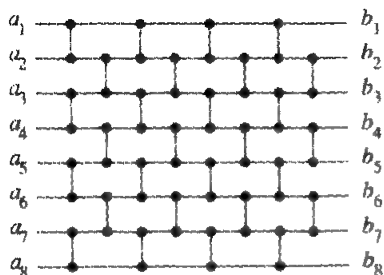
Một ***mạng sắp xếp chẵn-lẻ*** trên n đầu vào $(a_1, a_2, ..., a_n)$ có n cấp của các bộ so sánh. Hình 28.13 nêu một mạng chuyển vị chẵn-lẻ trên 8 đầu vào. Như có thể thấy trong hình, với $i = 2, 3, ..., n - 1$ và $d = 1, 2, ..., n$, dòng i được một bộ so sánh độ sâu-d nối với dòng $j = i + (-1)^{i+d}$ nếu $1 \le j \le n$.

c. Chứng minh họ của các mạng sắp xếp chẵn-lẻ quả thực là một họ các mạng sắp xếp.

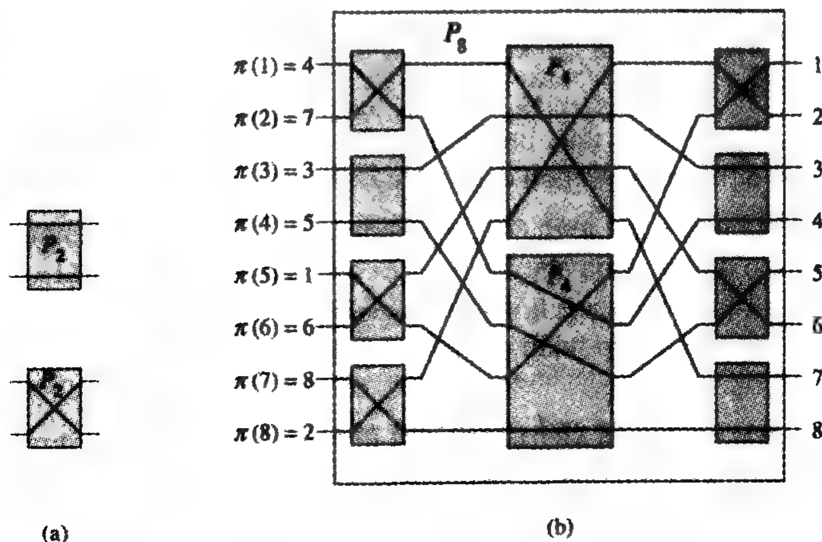
28-2 Mạng trộn chẵn-lẻ Batcher

Trong Đoạn 28.4, ta đã thấy cách kiến tạo một mạng trộn dựa trên kiểu sắp xếp bitonic. Trong bài toán này, ta sẽ kiến tạo một ***mạng trộn chẵn-lẻ***. Ta mặc nhận rằng n là một lũy thừa chính xác của 2, và ta muốn trộn dãy sắp xếp các thành phần trên các dòng $\langle a_1, a_2, ..., a_n\rangle$ với các thành phần trên các dòng $\langle a_{n+1}, a_{n+2}, ..., a_{2n}\rangle$. Ta kiến tạo đệ quy hai mạng trộn chẵn-lẻ sẽ trộn song song các dãy con đã sắp xếp. Mạng đầu tiên trộn dãy trên các dòng $\langle a_1, a_3, ..., a_{n-1}\rangle$ với dãy trên các dòng $\langle a_{n+1}, a_{n+3}, ..., a_{2n-1}\rangle$ (các thành phần lẻ). Mạng thứ hai trộn $\langle a_2, a_4, ..., a_n\rangle$ với $\langle a_{n+2}, a_{n+4}, ..., a_{2n}\rangle$ (các thành phần chẵn). Để tổ hợp hai dãy con đã sắp xếp, ta đặt một bộ so sánh giữa a_{2i-1} và a_{2i} với $i = 1, 2, ..., n$.

a. Vẽ một mạng trộn $2n$-đầu vào với $n = 4$.



Hình 28.13 Một mạng sắp xếp chẵn-lẻ trên 8 nhập liệu.



Hình 28.14 Các mạng hoán vị. **(a)** Mạng hoán vị P_2, bao gồm một khóa chuyển đơn có thể được ấn định theo một trong hai cách đã nêu. **(b)** Phần kiến tạo đệ quy của P_8 từ 8 khóa chuyển và hai P_4. Các khóa chuyển và các P_4 được ấn định để thực hiện phép hoán vị $\pi = \langle 4, 7, 3, 5, 1, 6, 8, 2 \rangle$.

b. Dùng nguyên lý zero-một để chứng minh mọi mạng trộn chẵn-lẻ $2n$-đầu vào đều là một mạng trộn.

c. Đâu là độ sâu của một mạng trộn chẵn-lẻ $2n$-đầu vào? Nêu kích cỡ của nó?

28-3 Các mạng hoán vị

Một ***mạng hoán vị*** trên n đầu vào và n kết xuất có các khóa chuyển cho phép nó nối các đầu vào với các kết xuất của nó theo bất kỳ trong số $n!$ phép hoán vị khả dĩ. Hình 28.14(a) nêu mạng hoán vị P_2 2-đầu vào, 2-kết xuất, bao gồm một khóa chuyển đơn có thể được ấn định để nạp các đầu vào của nó thẳng đến các kết xuất của nó hoặc đi qua chúng.

a. Chứng tỏ nếu ta thay mỗi bộ so sánh trong một mạng sắp xếp bằng khóa chuyển của Hình 28.14(a), mạng kết quả là một mạng hoán vị. Nghĩa là, với bất kỳ phép hoán vị π, ta có một cách để ấn định các khóa chuyển trong mạng sao cho đầu vào i được nối với kết xuất $\pi(i)$.

Hình 28.14(b) nêu phần kiến tạo đệ quy của một mạng hoán vị P_8 8-đầu vào, 8-kết xuất sử dụng hai bản sao của P_4 và 8 khóa chuyển. Các khóa chuyển đã được ấn định để thực hiện phép hoán vị π = <4, 7, 3, 5, 1, 6, 8, 2>, yêu cầu (theo đệ quy) P_4 đỉnh thực hiện <4, 2, 3, 1> và P_4 đáy thực hiện <2, 3, 1, 4>.

b. Nêu cách thực hiện phép hoán vị <5, 3, 4, 6, 1, 8, 2, 7> trên P_8 bằng cách vẽ các xác lập khóa chuyển và các phép hoán vị được thực hiện bởi hai P_4.

Cho n là một lũy thừa chính xác của 2. Định nghĩa P_n một cách đệ quy theo dạng hai $P_{n/2}$ theo cách tương tự như đã định nghĩa P_8.

c. Mô tả một thuật toán $O(n)$ thời gian (máy truy cập ngẫu nhiên bình thường) ấn định n khóa chuyển nối với các đầu vào và các kết xuất của P_n và chỉ định các phép hoán vị phải được thực hiện bởi từng $P_{n/2}$ để hoàn thành mọi phép hoán vị n-thành phần đã cho. Chứng minh thuật toán của bạn là đúng.

d. Đâu là độ sâu và kích cỡ của P_n? Phải mất bao lâu để một máy truy cập ngẫu nhiên bình thường tính toán tất cả xác lập khóa chuyển, kể cả trong các $P_{n/2}$?

e. Chứng tỏ với $n > 2$, bất kỳ mạng hoán vị nào—không chỉ P_n—đều phải thực hiện một phép hoán vị nào đó theo hai tổ hợp xác lập khóa chuyển riêng biệt.

Ghi chú Chương

Knuth [123] chứa một phần mô tả các mạng sắp xếp và lập biểu đồ lịch sử của chúng. Hình như chúng được khảo sát lần đầu tiên vào năm 1954 bởi P. N. Armstrong, R. J. Nelson, và D. J. O'Connor. Trong đầu những năm 1960, K. E. Batcher đã khám phá mạng đầu tiên có khả năng trộn hai dãy n con số trong $O(\lg n)$ thời gian. Ông đã dùng tiến trình trộn chẵn-lẻ (xem Bài toán 28-2), và cũng đã nêu cách dùng kỹ thuật này để sắp xếp n con số trong $O(\lg^2 n)$ thời gian. Không lâu sau đó, ông đã khám phá ra một bộ sắp xếp bitonic $O(\lg n)$-độ sâu tương tự như cái được trình bày trong Đoạn 28.3. Knuth quy nguyên lý zero-một cho W. G. Bouricius (1954), là người đã chứng minh nó trong ngữ cảnh các cây quyết định.

Trong một thời gian dài, một câu hỏi vẫn để mở đó là liệu có một mạng sắp xếp với độ sâu $O(\lg n)$ tồn tại hay không. Vào năm 1983, đáp án đã cho thấy là có song có phần không thỏa đáng. Mạng sắp xếp AKS (được đặt theo các nhà phát triển của nó, Ajtai, Komlos, và Szemeredi [8]) có thể sắp xếp n con số trong độ sâu $O(\lg n)$ dùng $O(n \lg n)$ bộ so sánh. Đáng tiếc, các hằng ẩn bởi hệ ký hiệu O khá lớn (hàng ngàn, ngàn), và như vậy nó không được xem là thực tiễn.

29 Các Mạch Số Học

Mô hình tính toán mà một máy tính bình thường cung cấp mặc nhận rằng các phép toán số học căn bản—cộng, trừ, nhân, và chia—có thể thực hiện trong thời gian bất biến. Cách nhìn tóm lược này là hợp lý, bởi các phép toán cơ bản nhất trên một máy truy cập ngẫu nhiên có các mức hao phí tương tự. Tuy nhiên, khi cần thiết kế hệ mạch thực thi các phép toán này, ta sớm phát hiện rằng khả năng thực hiện tùy thuộc vào tầm lớn của các con số đang được thao tác. Ví dụ, tất cả chúng ta đều đã học ở tiểu học cách cộng hai số tự nhiên, được diễn tả dưới dạng các số thập phân n-chữ số, trong $\Theta(n)$ bước (mặc dù thầy giáo thường không nhấn mạnh số bước cần có).

Chương này giới thiệu các mạch thực hiện các hàm số học. Với các tiến trình nối tiếp, $\Theta(n)$ là cận thời gian tiệm cận tốt nhất mà ta có thể hy vọng đạt được để cộng hai số n-chữ số. Tuy nhiên, với các mạch hoạt động song song, ta có thể làm tốt hơn. Trong chương này, ta sẽ thiết kế các mạch có thể nhanh chóng thực hiện phép cộng và phép nhân. (Phép trừ chủ yếu giống như phép cộng, và phép chia được để dành cho Bài toán 29-1.) Ta mặc nhận rằng tất cả các đầu vào là các số tự nhiên n-bit, được diễn tả theo nhị phân.

Để bắt đầu, Đoạn 29.1 trình bày các mạch tổ hợp. Ta sẽ xem độ sâu của một mạch tương ứng với "thời gian thực hiện" của nó ra sao. Bộ toàn cộng [full adder], là là một khối kiến tạo của hầu hết các mạch trong chương này, được dùng làm ví dụ đầu tiên của chúng ta về một mạch tổ hợp. Đoạn 29.2 trình bày hai mạch tổ hợp cho phép cộng: bộ cộng nhớ lược [ripple-carry adder], làm việc trong $\Theta(n)$ thời gian, và bộ cộng nhớ kiểm tra trước [carry-lookahead adder], chỉ mất $O(\lg n)$ thời gian. Nó cũng trình bày bộ cộng nhớ tiết kiệm [carry-save adder], có thể rút gọn bài toán lấy tổng ba con số thành bài toán lấy tổng hai con số trong $\Theta(1)$ thời gian. Đoạn 29.3 giới thiệu hai bộ nhân tổ hợp: bộ nhân mảng, mất $\Theta(n)$ thời gian, và bộ nhân cây Wallace, chỉ yêu cầu $\Theta(\lg n)$ thời gian. Cuối cùng, Đoạn 29.4 trình bày các mạch có các thành phần lưu trữ gắn đồng hồ (các thanh ghi) và nêu cách tiết kiệm phần cứng bằng cách dùng lại hệ mạch tổ hợp.

29.1 Các mạch tổ hợp

Cũng như các mạng so sánh ở Chương 28, các mạch tổ hợp hoạt động song song: nhiều thành phần có thể cùng lúc tính toán các giá trị dưới dạng một bước đơn. Trong đoạn này, ta định nghĩa các mạch tổ hợp và nghiên cứu cách tạo dựng các mạch tổ hợp lớn hơn từ các cổng cơ bản.

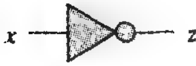
Các thành phần tổ hợp

Các mạch số học trong các máy tính thực được xây dựng từ các thành phần tổ hợp được tương kết bởi các dây dẫn. Một thành phần tổ hợp là một thành phần mạch bất kỳ có một số lượng bất biến các đầu vào và kết xuất và thực hiện một hàm được định nghĩa kỹ. Vài thành phần mà ta sẽ đề cập trong chương này là các thành phần tổ hợp bool—tất cả các đầu vào và kết xuất của chúng được rút từ tập hợp {0, 1}, ở đó 0 biểu diễn FALSE và 1 biểu diễn TRUE.

Một thành phần tổ hợp bool tính toán một hàm bool đơn giản được gọi là một cổng logic. Hình 29.1 nêu bốn cổng logic cơ bản sẽ được dùng làm các thành phần tổ hợp trong chương này: cổng ***NOT*** (hoặc cổng đảo), cổng ***AND***, cổng ***OR***, và cổng ***XOR***. (Nó cũng nêu hai cổng logic—cổng ***NAND*** và cổng ***NOR***—mà một số bài tập yêu cầu.) Cổng NOT nhận một đầu vào nhị phân x đơn giản, có giá trị là 0 hoặc 1, và tạo ra một kết xuất nhị phân z có giá trị trái ngược với giá trị đầu vào. Mỗi trong ba cổng khác nhận hai đầu vào nhị phân x và y và tạo ra một kết xuất nhị phân đơn z.

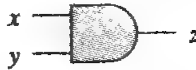
Phép toán của mỗi cổng, và của bất kỳ thành phần tổ hợp bool nào, có thể được mô tả bởi một bảng thực [truth table], được nêu dưới mỗi cổng trong Hình 29.1. Bảng thực cung cấp các kết xuất của thành phần tổ hợp cho mỗi xác lập khả dĩ của các đầu vào. Ví dụ, bảng thực của cổng XOR cho ta biết khi các đầu vào là $x = 0$ và $y = 1$, giá trị kết xuất sẽ là $z = 1$; nó tính toán "OR loại trừ" của hai đầu vào của nó. Ta dùng các ký hiệu $\neg$ để thể hiện hàm NOT, $\wedge$ để thể hiện hàm AND, $\vee$ để thể hiện hàm OR, và $\oplus$ để thể hiện hàm XOR. Do đó, ví dụ, $0 \oplus 1 = 1$.

Các thành phần tổ hợp trong các mạch thực không hoạt động tức thời. Sau khi các giá trị đầu vào nhập vào một thành phần tổ hợp ổn định, hoặc trở thành ổn định—nghĩa là, giữ ổn định trong một thời gian đủ lâu—giá trị kết xuất của thành phần được bảo đảm trở thành ổn định lẫn đúng đắn trong một thời lượng cố định sau đó. Ta gọi sự chênh lệch thời gian này là độ trễ lan truyền của thành phần. Chương này mặc nhận tất cả các thành phần tổ hợp đều có độ trễ lan truyền bất biến.



x	$\neg x$
0	1
1	0

(a)



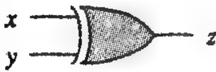
x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

(b)



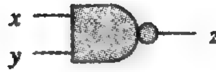
x	y	$x \vee y$
0	0	0
[illegible]	1	1
1	0	1
1	1	1

(c)



x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

(d)



x	y	$\neg(x \wedge y)$
0	0	1
0	1	1
1	0	1
1	1	0

(e)



x	y	$\neg(x \vee y)$
0	0	1
0	1	0
1	0	0
1	1	0

(f)

Hình 29.1 Sáu cổng logic cơ bản, có các đầu vào và kết xuất nhị phân. Dưới mỗi cổng là bảng thực mô tả phép toán của cổng. **(a)** Cổng NOT. **(b)** Cổng AND. **(c)** Cổng OR. **(d)** Cổng XOR (OR loại trừ). **(e)** Cổng NAND (NOT-AND). **(f)** Cổng NOR (NOT-OR).

Các mạch tổ hợp

Một *mạch tổ hợp* bao gồm một hoặc nhiều thành phần tổ hợp tương kết theo kiểu phi chu trình. Các tuyến tương kết được gọi là các ***dây dẫn*** [wires]. Một dây dẫn có thể nối kết xuất của một thành phần với đầu vào của một thành phần khác, và do đó cung cấp giá trị kết xuất của thành phần đầu tiên làm giá trị đầu vào của thành phần thứ hai. Tuy một dây dẫn đơn có thể không có nhiều hơn một kết xuất thành phần tổ hợp nối với nó, song nó có thể nạp vài đầu vào thành phần. Số lượng các đầu vào thành phần mà một dây dẫn nạp được gọi là ***mức lùa ra*** [fan-out] của dây dẫn. Nếu không có kết xuất thành phần nào nối với một dây dẫn, thì dây dẫn đó là một ***mạch đầu vào***, chấp nhận các giá trị đầu vào từ một nguồn bên ngoài. Nếu không có đầu vào thành phần nào nối với một dây dẫn, thì dây dẫn đó là một ***mạch kết xuất***, cung cấp các kết quả tính toán của mạch cho thế giới bên ngoài. (Một dây dẫn bên trong cũng có thể "lùa ra" [fan out] đến một mạch kết xuất.) Các mạch tổ hợp không chứa các chu trình và không có các thành phần bộ nhớ (như các thanh ghi mô tả trong Đoạn 29.4).

Các bộ toàn cộng

Để lấy ví dụ, Hình 29.2 nêu một mạch tổ hợp, có tên ***bộ toàn cộng*** [full adder], nhận ba bit x, y, và z làm đầu vào. Nó kết xuất hai bit, s và c, theo bảng thực dưới đây:

x	y	z	c	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Kết xuất s là ***tính chẵn lẻ*** [parity]của các bit đầu vào,

$$s = \text{tính chẵn lẻ}(x, y, z) = x \oplus y \oplus z \,, \qquad (29.1)$$

và kết xuất c là ***phần lớn*** [majority] của các bit đầu vào,

$$c = \text{phần lớn}(x, y, z) = (x \wedge y) \vee (y \wedge z) \vee (x \wedge z) \,. \qquad (29.2)$$

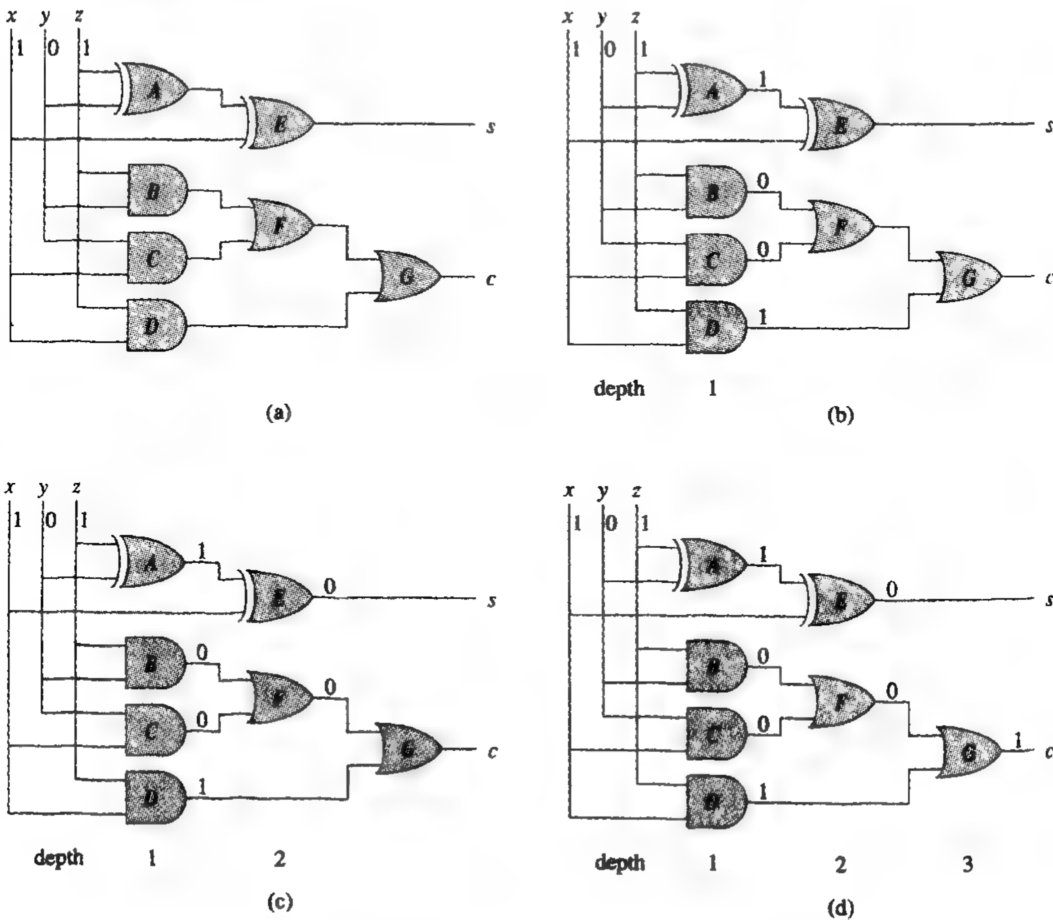
(Nói chung, hàm tính chẵn lẻ và hàm phần lớn có thể chấp nhận bất kỳ số lượng bit đầu vào nào. Tính chẵn lẻ là 1 nếu và chỉ nếu một số lượng lẻ các đầu vào là các 1. Phần lớn là 1 nếu và chỉ nếu trên nửa số đầu vào là các 1.) Lưu ý, các bit c và s, chung với nhau, cho ra tổng của x, y, và z. Ví dụ, nếu $x = 1$, $y = 0$, và $z = 1$, thì $\langle c, s\rangle = \langle 10\rangle$,[1] là phần biểu diễn nhị phân của 2, tổng của x, y, và z.

Mỗi đầu vào x, y, và z cho bộ toàn cộng có một mức lùa ra là 3. Khi phép toán được thực hiện bởi một thành phần tổ hợp là giao hoán và kết hợp đối với các đầu vào của nó (như các hàm AND, OR, và XOR), ta gọi số lượng đầu vào là ***mức lùa vào*** [fan-in] của thành phần. Tuy mức lùa vào của mỗi cổng trong Hình 29.2 là 2, ta vẫn có thể vẽ lại bộ toàn cộng để thay các cổng XOR A và E bằng một cổng XOR đơn 3 - đầu vào và các cổng OR F và G bằng một cổng OR đơn 3 - nhập liệu.

Để xem xét cách hoạt động của bộ toàn cộng, ta mặc nhận mỗi cổng hoạt động trong thời gian đơn vị. Hình 29.2(a) nêu một loạt các đầu vào

[1] Để minh bạch, ta bỏ qua các dấu phẩy giữa các thành phần của dãy khi chúng là các bit.

trở thành ổn định vào lúc 0. Các cổng *A-D*, và không có cổng nào khác, có tất cả các giá trị đầu vào của chúng ổn định vào lúc đó và do đó tạo ra các giá trị trong hình 29.2(b) vào lúc 1. Lưu ý, các cổng *A-D* hoạt động song song, các cổng *E* và *F*, chứ không phải cổng *G*, có các đầu vào ổn định vào lúc 1 và tạo các giá trị nêu trong Hình 29.2(c) vào lúc 2. Kết xuất của cổng *E* là bit *s*, và do đó kết xuất *s* từ bộ toàn cộng đã sẵn sàng vào lúc 2. Tuy nhiên, kết xuất *c* chưa sẵn sàng. Cổng *G* cuối cùng có các đầu vào ổn định vào lúc 2, và nó tạo kết xuất *c* nêu trong Hình 29.2(d) vào lúc 3.



Hình 29.2 Một mạch toàn cộng. **(a)** Vào lúc 0, các bit đầu vào đã nêu sẽ xuất hiện trên ba dây dẫn đầu vào. **(b)** Vào lúc 1, các giá trị đã nêu sẽ xuất hiện trên các kết xuất của các cổng *A-D*, tại độ sâu 1. **(c)** Vào lúc 2, các giá trị đã nêu sẽ xuất hiện trên các kết xuất của các cổng *E* và F, tại độ sâu 2. **(d)** Vào lúc 3, cổng *G* tạo ra kết xuất của nó, cũng là mạch kết xuất.

Độ sâu mạch

Như trong trường hợp các mạng so sánh mô tả trong Chương 28, ta đo độ trễ lan truyền của một mạch tổ hợp theo dạng số lượng lớn nhất của các thành phần tổ hợp trên một lộ trình bất kỳ từ các đầu vào đến các kết xuất. Cụ thể, ta định nghĩa ***độ sâu*** của một mạch, tương ứng với "thời gian thực hiện" trường hợp xấu nhất của nó, mà theo quy nạp có dạng các độ sâu của các dây dẫn thành phần của nó. Độ sâu của một dây dẫn đầu vào là 0. Nếu một thành phần tổ hợp có các đầu vào x_1, $x_2, \ldots, x_n$ tại các độ sâu $d_1, d_2, \ldots, d_n$ theo thứ tự nêu trên, thì các kết xuất của nó có độ sâu $\max\{d_1, d_2, \ldots, d_n\} + 1$. Độ sâu của một thành phần tổ hợp là độ sâu các kết xuất của nó. Độ sâu của một mạch tổ hợp là độ sâu cực đại của bất kỳ thành phần tổ hợp nào. Bởi ta cấm các mạch tổ hợp chứa các chu trình, nên nhiều khái niệm khác nhau về độ sâu được định nghĩa kỹ.

Nếu mỗi thành phần tổ hợp bỏ ra một thời gian bất biến để tính toán các giá trị kết xuất của nó, thì độ trễ lan truyền trường hợp xấu nhất qua một mạch tổ hợp sẽ tỷ lệ với độ sâu của nó. Hình 29.2 nêu độ sâu của mỗi cổng trong bộ toàn cộng. Bởi cổng có độ sâu lớn nhất là cổng G, nên chính bộ toàn cộng có độ sâu 3, tỷ lệ với thời gian trường hợp xấu nhất mà nó trải qua để mạch thực hiện chức năng của nó.

Đôi lúc một mạch tổ hợp có thể tính toán nhanh hơn độ sâu của nó. Giả sử một mạch con lớn nạp vào một đầu vào của một cổng AND 2-đầu vào nhưng giả sử rằng đầu vào kia của cổng AND có giá trị 0. Như vậy, kết xuất của cổng sẽ là 0, độc lập với đầu vào từ mạch con lớn. Tuy nhiên, nói chung, ta không thể tin chắc vào các đầu vào cụ thể đang được áp dụng cho mạch, và do đó việc trừu tượng hóa độ sâu dưới dạng "thời gian thực hiện" của mạch khá hợp lý.

Kích cỡ mạch

Ngoài độ sâu mạch, có một tài nguyên khác mà ta thường muốn giảm thiểu khi thiết kế các mạch. Kích cỡ của một mạch tổ hợp là số lượng các thành phần tổ hợp mà nó chứa. Theo trực giác, kích cỡ mạch tương ứng với không gian bộ nhớ mà một thuật toán sử dụng. Ví dụ, bộ toàn cộng của Hình 29.2 có kích cỡ 7, bởi nó sử dụng 7 cổng.

Định nghĩa này về kích cỡ mạch không thật hữu ích đối với các mạch nhỏ. Xét cho cùng, do một bộ toàn cộng có một số lượng đầu vào và kết xuất không đổi và tính toán một hàm được định nghĩa kỹ, nên nó thỏa phần định nghĩa của một thành phần tổ hợp. Do đó, một bộ toàn cộng được xây dựng từ một thành phần tổ hợp bộ toàn cộng đơn sẽ có kích cỡ 1. Thực vậy, theo theo định nghĩa này, *mọi* thành phần tổ hợp

đều có kích cỡ 1.

Phần định nghĩa kích cỡ mạch chủ yếu được áp dụng cho các họ mạch tính toán các hàm tương tự. Ví dụ, ta sẽ sớm thấy một mạch cộng tiếp nhận hai đầu vào *n*-bit. Thực tế ở đây ta không nói về một mạch đơn, nhưng thay vì thế là một họ mạch—mỗi mạch cho một kích cỡ đầu vào.

Trong ngữ cảnh này, phần định nghĩa kích cỡ mạch rất có ý nghĩa. Nó cho phép ta định nghĩa các thành phần mạch tiện dụng mà không làm ảnh hưởng đến kích cỡ của bất kỳ kiểu thực thi nào của mạch theo nhiều hơn một thừa số bất biến. Tất nhiên, trong thực tế, các phép đo kích cỡ thường phức tạp hơn nhiều, bao hàm không những sự chọn lựa các thành phần tổ hợp, mà còn liên quan đến những thứ như diện tích mà mạch yêu cầu khi tích hợp trên một chip silicon.

Bài tập

29.1-1

Trong Hình 29.2, thay đổi đầu vào *y* thành một 1. Nêu giá trị kết quả được mang tải trên mỗi dây dẫn.

29.1-2

Nêu cách kiến tạo một mạch chẵn lẻ *n*-đầu vào có $n - 1$ cổng XOR và độ sâu $\lceil \lg n \rceil$.

29.1-3

Chứng tỏ có thể xây dựng một thành phần tổ hợp bool bất kỳ từ một lượng bất biến các cổng AND, OR, và NOT. (*Mách nước:* Thực thi bảng thực cho thành phần đó.)

29.1-4

Chứng tỏ có thể hoàn toàn xây dựng một hàm bool bất kỳ từ các cổng NAND.

29.1-5

Kiến tạo một mạch tổ hợp thực hiện hàm OR loại trừ chỉ dùng bốn cổng NAND 2-đầu vào.

29.1-6

Cho *C* là một mạch tổ hợp *n*-đầu vào, *n*-kết xuất có độ sâu *d*. Nếu hai bản sao *C* được nối, với các kết xuất của bản sao này nạp trực tiếp vào các đầu vào của bản sao kia, nêu độ sâu khả dĩ cực đại của mạch nối đuôi này? Nêu độ sâu khả dĩ cực tiểu?

29.2 Các mạch cộng

Giờ đây, ta nghiên cứu bài toán cộng các con số được biểu diễn theo nhị phân. Ta trình bày ba mạch tổ hợp cho bài toán này. Thứ nhất, ta xét phép cộng nhớ lược, có thể cộng hai con số n-bit trong $\Theta(n)$ thời gian dùng một mạch có kích cỡ $\Theta(n)$. Để cải thiện cận thời gian này theo $O(\lg n)$, ta dùng một bộ cộng nhớ kiểm tra trước, cũng có kích cỡ $\Theta(n)$. Cuối cùng, ta trình bày phép cộng nhớ tiết kiệm, mà trong $O(1)$ thời gian nó có thể rút gọn tổng của 3 số n-bit thành tổng của một số n-bit và một số $(n + 1)$ bit. Mạch có kích cỡ $\Theta(n)$.

8	7	6	5	4	3	2	1	0		i
1	1	0	1	1	1	0	0	0	=	c
	0	1	0	1	1	1	1	0	=	a
	1	1	0	1	0	1	0	1	=	b
1	0	0	1	1	0	0	1	1	=	s

Hình 29.3 Bổ sung hai số 8-bit a = <01011110> và b = <11010101> để tạo ra một tổng 9-bit s = <100110011>. Mỗi bit c_i là một bit nhớ. Mỗi cột bit biểu diễn, từ trên xuống, c_i, a_i, b_i, và s_i với một i. c_0 nhớ-trong luôn là 0.

29.2.1 Phép cộng nhớ lược

Ta bắt đầu với phương pháp lấy tổng các con số nhị phân bình thường. Mặc nhận rằng một số nguyên không âm a được biểu thị theo nhị phân bằng một dãy n bit $\langle a_{n-1}, a_{n-2}, \ldots, a_0 \rangle$, ở đó $n \le \lceil \lg(a + 1) \rceil$ và

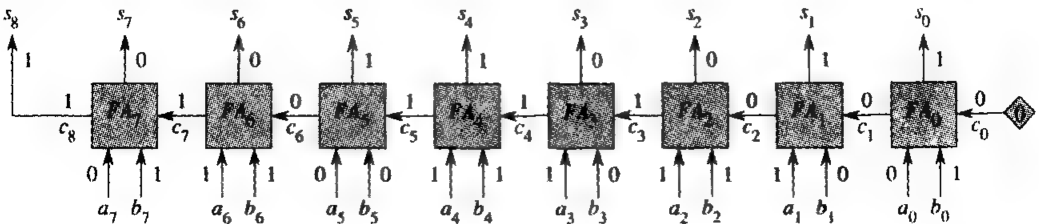
$$a = \sum_{i=0}^{n-1} a_i 2^i .$$

Cho hai số n-bit $a = \langle a_{n-1}, a_{n-2}, \ldots, a_0 \rangle$ và $b = \langle b_{n-1}, b_{n-2}, \ldots, b_0 \rangle$, ta muốn tạo ra một tổng $(n + 1)$ bit $s = \langle s_n, s_{n-1}, \ldots, s_0 \rangle$. Hình 29.3 nêu một ví dụ về cách cộng hai số 8-bit. Ta lấy tổng các cột từ phải qua trái, lan truyền mọi trường hợp nhớ từ cột i sang cột $i + 1$, với $i = 0, 1, \ldots, n - 1$. Tại vị trí bit thứ i, ta sử dụng các bit a_i và b_i và một ***bit nhớ-trong*** c_i làm đầu vào, rồi tạo một ***bit tổng*** s_i và một ***bit nhớ-ngoài*** c_{i+1}. Bit nhớ - ngoài c_{i+1} từ vị trí thứ i là bit nhớ-trong vào vị trí thứ $(i + 1)$. Bởi không có một trường hợp nhớ - trong nào cho vị trí 0, nên ta mặc nhận rằng $c_0 = 0$. c_n nhớ-ngoài là bit s_n của tổng.

Nhận thấy mỗi bit tổng s_i là tính chẵn lẻ của các bit a_i, b_i, và c_i (xem phương trình (29.1)). Hơn nữa, bit nhớ-ngoài c_{i+1} là phần lớn của a_i, b_i, và c_i (xem phương trình (29.2)). Như vậy, mỗi giai đoạn của phép cộng có thể được thực hiện bằng một bộ toàn cộng.

Một ***bộ cộng nhớ lược*** n-bit được hình thành bằng cách liên hoàn [cascading] n bộ toàn cộng FA_0, FA_1,..., FA_{n-1}, nạp c_{i+1} nhớ-ngoài của FA_i trực tiếp vào đầu vào nhớ-trong của FA_{i+1}. Hình 29.4 nêu một bộ cộng nhớ lược 8-bit. Các bit nhớ sẽ "lược" [ripple] từ phải qua trái. c_0 nhớ-trong với bộ toàn cộng FA_1 được ***đấu chết*** [hardwired] với 0, nghĩa là, nó là 0 bất kể các giá trị mà các đầu vào khác đảm nhiệm. Kết xuất là số $(n + 1)$ bit $s = <s_n, s_{n-1},..., s_0>$, ở đó s_n bằng c_n, bit nhớ-ngoài từ bộ toàn cộng FA_n.

Bởi các bit nhớ lược qua tất cả n bộ toàn cộng, nên thời gian mà một bộ cộng nhớ lược n-bit yêu cầu là $\Theta(n)$. Một cách chính xác hơn, bộ toàn cộng FA_i nằm tại độ sâu $i + 1$ trong mạch. Bởi FA_{n-1} nằm tại độ sâu lớn nhất của bất kỳ bộ toàn cộng nào trong mạch, nên độ sâu của bộ cộng nhớ lược là n. Kích cỡ của mạch là $\Theta(n)$ bởi nó chứa n thành phần tổ hợp.



Hình 29.4 Một bộ cộng nhớ lược 8-bit thực hiện phép cộng của Hình 29.3. Bit nhớ c_0 được đấu chết với 0, được nêu rõ bằng hình thoi, và các bit nhớ sẽ lược từ phải qua trái.

29.2.2 Phép cộng nhớ kiểm tra trước

Phép cộng nhớ lược yêu cầu $\Theta(n)$ thời gian bởi tiến trình lược của các bit nhớ qua mạch. Phép cộng nhớ kiểm tra trước tránh độ trễ $\Theta(n)$ thời gian này bằng cách gia tốc việc tính toán các trường hợp nhớ dùng một mạch có hình cây. Một bộ cộng nhớ kiểm tra trước có thể lấy tổng hai số n-bit trong $O(\lg n)$ thời gian.

Nhận xét chính đó là trong phép cộng nhớ lược, với $i \geq 1$, bộ toàn cộng FA_i có hai trong số các giá trị đầu vào của nó, tức a_i và b_i, đã sẵn sàng từ lâu trước khi c_i nhớ-trong sẵn sàng. Ý tưởng đàng sau bộ cộng nhớ kiểm tra trước đó là khai thác thông tin từng phần này.

Để lấy ví dụ, cho $a_{i-1} = b_{i-1}$. Bởi c_i nhớ-ngoài là hàm phần lớn, ta có $c_i = a_{i-1} = b_{i-1}$, *bất chấp* c_{i-1}, *nhớ-trong*. Nếu $a_{i-1} = b_{i-1} = 0$, ta có thể ***triệt*** [kill] c_i nhớ-ngoài bằng cách buộc nó theo 0 mà không đợi giá trị của c_{i-1}, được tính toán. Cũng vậy, nếu $a_{i-1} = b_{i-1} = 1$, ta có thể ***phát sinh*** c_i

= 1 nhớ-ngoài, bất luận giá trị của c_{i-1}.

Tuy nhiên, nếu $a_{i-1} \neq b_{i-1}$, thì c_i tùy thuộc vào c_{i-1}. Cụ thể, $c_i = c_{i-1}$, bởi c_{i-1} nhớ-trong áp đổi "lá phiếu" quyết định trong cuộc bầu phần lớn để xác định c_i. Trong trường hợp này, ta ***lan truyền*** phần nhớ, bởi phần nhớ-ngoài là nhớ-trong.

Hình 29.5 tóm tắt các mối quan hệ này theo dạng các ***tình trạng nhớ***, ở đó *k* là "triệt nhớ," g là "phát sinh nhớ," và *p* là "lan truyền nhớ."

Xét chung hai bộ toàn cộng liên tiếp FA_{i-1}, và FA_i với nhau dưới dạng một đơn vị phối hợp. Phần nhớ-trong cho đơn vị là c_{i-1}, và phần nhớ-ngoài là c_{i+1}. Ta có thể xem đơn vị phối hợp như là triệt, phát sinh, hoặc lan truyền các trường hợp nhớ, tương tự như một bộ toàn cộng đơn. Đơn vị phối hợp sẽ triệt trường hợp nhớ củanó nếu FA_i triệt trường hợp nhớ củanó hoặc nếu FA_{i-1}, triệt trường hợp nhớ của nó và FA_i lan truyền nó. Cũng vậy, đơn vị phối hợp phát sinh một trường hợp nhớ nếu FA_i phát sinh một trường hợp nhớ hoặc nếu FA_{i-1}, phát sinh một trường hợp nhớ và FA_i lan truyền nó. Đơn vị phối hợp lan truyền trường hợp nhớ, ấn định $c_{i+1} = c_{i-1}$, nếu cả hai bộ toàn cộng lan truyền các trường hợp nhớ. Bảng trong Hình 29.6 tóm lược cách phối hợp các tình trạng nhớ khi các bộ toàn cộngkề nhau. Ta có thể xem bảng này như là phần định nghĩa của ***toán tử tình trạng nhớ*** [carry-status operator] ⊗ trên miền {k, p, g}. Một tính chất quan trọng của toán tử này đó là nó mang tính kết hợp [associative], như Bài tập 29.2-2 yêu cầu bạn xác minh.

a_{i-1}	b_{i-1}	c_i	tình trạng nhớ
0	0	0	k
0	1	c_{i-1}	g
1	0	c_{i-1}	p
1	1	1	g

Hình 29.5 Bit nhớ-ngoài c_i và tình trạng nhớ tương ứng với các đầu vào a_{i-1}, b_{i-1}, và c_{i-1}, của bộ toàn cộng FA_{i-1}, trong phép cộng nhớ lược.

		FA_i		
	⊗	k	p	g
	k	k	k	g
FA_{i-1}	p	k	p	g
	g	k	g	g

Hình 29.6 Tình trạng nhớ của tổ hợp các bộ toàn cộng FA_{i-1}, và FA_i theo dạng các tình trạng nhớ riêng lẻ của chúng, căn cứ vào toán tử tình trạng nhớ ⊗ trên miền {k, p, g}.

Ta có thể dùng toán tử tình trạng nhớ để diễn tả mỗi bit nhớ c_i theo dạng các đầu vào. Ta bắt đầu bằng cách định nghĩa $x_0 = k$ và

$$x_i = \begin{cases} k & \text{nếu } a_{i-1} = b_{i-1} = 0, \\ p & \text{nếu } a_{i-1} \neq b_{i-1}, \\ g & \text{nếu } a_{i-1} = b_{i-1} = 1, \end{cases} \qquad (29.3)$$

với $i = 1, 2, ..., n$. Như vậy, với $i = 1, 2, ..., n$, giá trị của x_i là tình trạng nhớ căn cứ vào Hình 29.5.

c_i nhớ-ngoài của một bộ toàn cộng đã cho FA_{i-1}, có thể tùy thuộc vào tình trạng nhớ của mọi bộ toàn cộng FA_j với $j = 0, 1, ..., i - 1$. Ta hãy định nghĩa $y_0 = x_0 = k$ và

$$\begin{aligned} y_i &= y_{i-1} \otimes x_i \qquad (29.4) \\ &= x_0 \otimes x_1 \otimes ... \otimes x_i \end{aligned}$$

với $i = 1, 2, ..., n$. Ta có thể xem *yi* như một "tiền tố" của $x_0 \otimes x_1 \otimes ... \otimes x_n$; ta gọi tiến trình tính toán các giá trị $y_0, y_1, ..., y_n$ là một ***phép tính tiền tố***. (Chương 30 sẽ mô tả các phép tính tiền tố trong ngữ cảnh song song mang tính chung hơn.) Hình 29.7 nêu các giá trị của x_i và y_i tương ứng với phép cộng nhị phân nêu trong Hình 29.3. Bổ đề dưới đây nêu ý nghĩa của các giá trị y_i cho phép cộng nhớ kiểm tra trước.

i	8	7	6	5	4	3	2	1	0
a_i		0	1	0	1	1	1	1	0
b_i		1	1	0	1	0	1	0	1
x_i	p	g	k	g	p	g	p	p	k
y_i	g	g	k	g	g	g	k	k	k
c_i	1	1	0	1	1	1	0	0	0

Hình 29.7 Các giá trị của x_i và y_i với $i = 0, 1, ..., 8$ tương ứng với các giá trị của a_i, b_i, và c_i trong bài toán phép cộng nhị phân của Hình 29.3. Mỗi giá trị của x_i được tô bóng với các giá trị của a_{i-1}, và b_{i-1}, mà nó tùy thuộc vào.

Bổ đề 29.1

Định nghĩa $x_0, x_1, ..., x_n$ và $y_0, y_1, ..., y_n$ theo các phương trình (29.3) và (29.4). Với $i = 0, 1, ..., n$, các điều kiện dưới đây tồn tại:

1. y_i = **k** hàm ý $c_i = 0$,

2. y_i = **g** hàm ý $c_i = 1$, và

3. y_i = **p** không xảy ra.

Chứng minh Phần chứng minh theo phương pháp quy nạp trên i. Về cơ bản, $i = 0$. Ta có $y_0 = x_0 = k$ theo định nghĩa, và cũng $c_0 = 0$. Với bước quy nạp, ta mặc nhận rằng bổ đề đứng vững với i - 1. Có ba trường hợp tùy thuộc vào giá trị của y_i.

1. Nếu $y_i = k$, thì do $y_i = y_{i-1} \otimes x_i$, phần định nghĩa của toán tử tình trạng nhớ ⊗ từ Hình 29.6 hàm ý rằng x_i = **k** hoặc x_i = **p** và y_{i-1} = **k**. Nếu x_i = **k**, thì phương trình (29.3) hàm ý rằng $a_{i-1} = b_{i-1} = 0$, và như vậy c_i = majority(a_{i-1}, b_{i-1}, c_{i-1}) = 0. Nếu x_i = **p** và y_{i-1} = **k**, thì $a_{i-1} \neq b_{i-1}$ và, theo phương pháp quy nạp, $c_{i-1} = 0$. Như vậy, majority(a_{i-1}, b_{i-1}, c_{i-1}) = 0, và như vậy $c_i = 0$.

2. Nếu y_i = **g**, thì ta có x_i = **g** hoặc có x_i = **p** và y_{i-1} = **g**. Nếu x_i = **g**, thì $a_{i-1} = b_{i-1} = 1$, hàm ý $c_i = 1$. Nếu x_i = **p** và y_{i-1} = **g**, thì $a_{i-1} \neq b_{i-1}$ và, theo phương pháp quy nạp, $c_{i-1} = 1$, hàm ý $c_i = 1$.

3. Nếu y_i = **p**, thì Hình 29.6 hàm ý y_{i-1} = **p**, mâu thuẫn với giả thuyết quy nạp.

Bổ đề 29.1 hàm ý rằng ta có thể tính toán từng bit nhớ c_i bằng cách tính toán từng tình trạng nhớ y_i. Sau khi có tất cả các bit nhớ, ta có thể tính toán nguyên cả tổng trong Θ(1) thời gian bằng cách tính toán song song các bit tổng s_i = parity(a_i, b_i, c_i) với i = 0, 1,..., n (lấy $a_n = b_n = 0$). Như vậy, bài toán cộng nhanh hai con số sẽ rút gọn thành phép tính tiền tố của các tình trạng nhớ $y_0, y_1, \ldots, y_n$.

Tính toán các tình trạng nhớ với một mạch tiền tố song song

Nhờ dùng một mạch tiền tố vận hành song song, trái với một mạch nhớ lược [ripple-carry] lần lượt tạo ra từng kết xuất một của nó, ta có thể tính toán tất cả n tình trạng nhớ $y_0, y_1, \ldots, y_n$ nhanh chóng hơn. Cụ thể, ta sẽ thiết kế một mạch tiền tố song song với độ sâu $O(\lg n)$. Mạch có kích cỡ $\Theta(n)$—theo tiệm cận có cùng lượng phần cứng như bộ cộng nhớ lược.

Trước khi kiến tạo mạch tiền tố song song, ta giới thiệu qua một hệ ký hiệu sẽ giúp ta hiểu cách vận hành của mạch. Với các số nguyên i và j trong miền giá trị $0 \leq i \leq j \leq n$, ta định nghĩa

$$[i, j] = x_i \otimes x_{i+1} \otimes \ldots \otimes x_j .$$

Như vậy, với $i = 0, 1, \ldots, n$, ta có $[i, i] = x_i$, bởi sự cấu thành chỉ một tình trạng nhớ x_i lại chính là nó. Với i, j, và k thỏa $0 \leq i \leq j \leq k \leq n$, ta cũng có đồng nhất thức

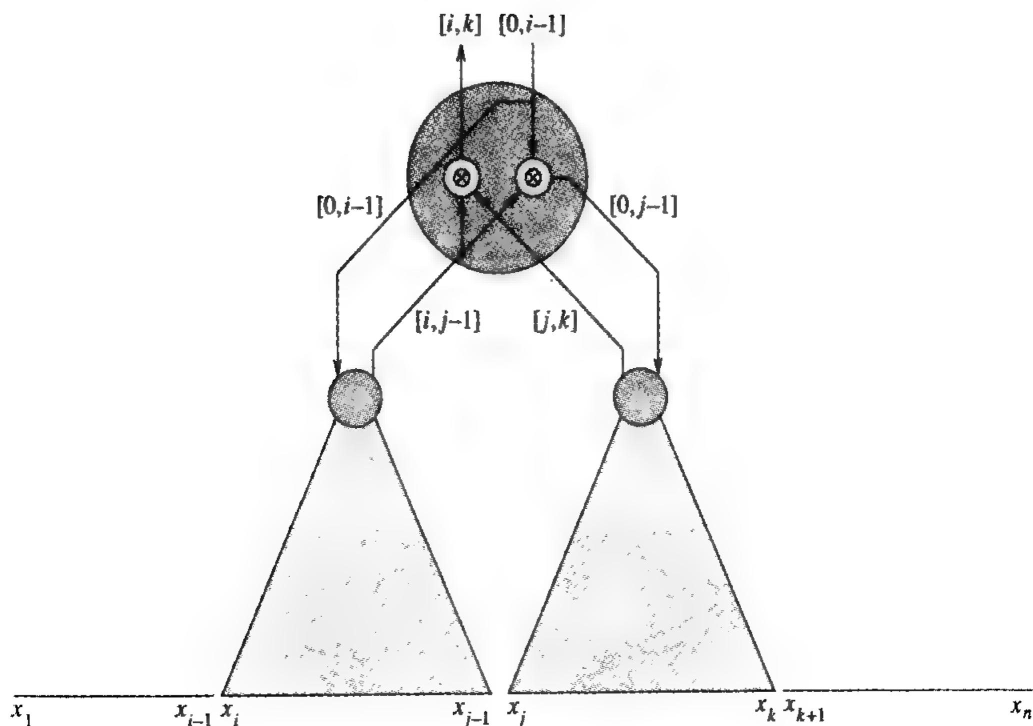
$$[i, k] = [i, j-1] \otimes [j, k], \qquad (29.5)$$

bởi toán tử tình trạng nhớ mang tính kết hợp. Mục tiêu của một phép tính tiền tố, theo dạng hệ ký hiệu này, đó là tính toán $y_i = [0, i]$ với $i = 0, 1, \ldots, n$.

Thành phần tổ hợp duy nhất được dùng trong mạch tiền tố song song là một mạch tính toán toán tử ⊗. Hình 29.8 nêu cách tổ chức các cặp ⊗ thành phần để tạo thành các nút trong của một cây nhị phân hoàn chỉnh, và Hình 29.9 minh họa mạch tiền tố song song với $n = 8$. Lưu ý, các dây dẫn trong mạch theo cấu trúc của một cây, nhưng bản thân mạch không phải là một cây, mặc dù nó thuần túy mang tính tổ hợp. Các đầu vào x_1, $x_2, \ldots, x_n$ được cung cấp tại các lá, và đầu vào x_0 được cung cấp tại gốc. Các kết xuất $y_0, y_1, \ldots, y_{n-1}$ được tạo tại các lá, và kết xuất yæ được tạo tại gốc. (Để tạo thuận lợi cho việc tìm hiểu phép tính tiền tố, các chỉ số biến gia tăng từ trái qua phải trong các Hình 29.8 và 29.9, thay vì từ phải qua trái như trong các hình khác của đoạn này.)

Hai thành phần ⊗ trong mỗi nút thường hoạt động vào các thời điểm khác nhau và có các độ sâu khác nhau trong mạch. Như đã nêu trong Hình 29.8, nếu cây con có gốc tại một nút đã cho trải rộng vài miền x_i, $x_{i+1}, \ldots, x_k$ của các đầu vào, cây con trái của nó trải rộng miền $x_i, x_{i+1}, \ldots, x_{j-1}$, và cây con phải của nó trải rộng miền $x_j, x_{j+1}, \ldots, x_k$, như vậy nút buộc phải tạo ra cho cha của nó tích $[i, k]$ của tất cả các đầu vào mà cây con của nó trải rộng. Bởi theo quy nạp ta có thể mặc nhận các con trái và phải của nút tạo ra các tích $[i, j-1]$ và $[j, k]$, nút đơn giản sử dụng một trong hai thành phần của nó để tính toán $[i, k] \leftarrow [i, j-1] \otimes [j, k]$.

Vào một lúc nào đó sau khi giai đoạn đi lên này của phép tính, nút sẽ nhận từ cha của nó tích $[0, i-1]$ của tất cả các đầu vào đến trước đầu vào nút trái x_i mà nó trải rộng. Tương tự như vậy, giờ đây nút tính toán các giá trị cho các con của nó. Đầu vào nút trái mà con trái của nút trải rộng cũng là x_i, và do đó nó chuyền giá trị $[0, i-1]$ cho con trái không thay đổi. Đầu vào nút trái mà con phải của nó trải rộng là x_j, và do đó nó phải tạo ra $[0, j-1]$.

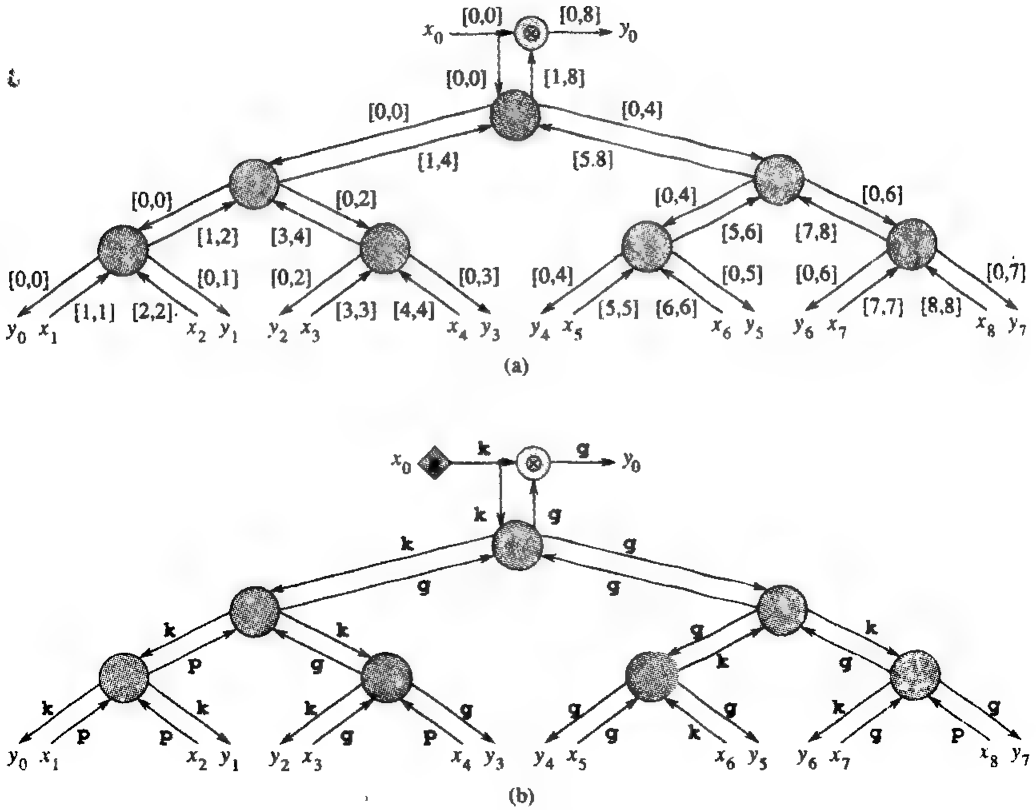


Hình 29.8 Tổ chức của một mạch tiền tố song song. Mắt đã nêu là gốc của một cây con có các lá nhập các giá trị x_i đến x_k. Cây con trái của mắt trải rộng các đầu vào x_i đến x_{j-1}, và cây con phải của nó trải rộng các đầu vào x_j đến x_k. Mắt bao gồm hai thành phần ⊗, hoạt động tại các thời điểm khác nhau trong phép toán của mạch. Một thành phần tính toán $[i, k] \leftarrow [i, j-1] \otimes [j, f]$, và thành phần kia tính toán $[0, j-1] - [0, i-1] \otimes [i, j-1]$. Các giá trị đã tính toán được nêu trên các dây dẫn.

Do nút nhận giá trị $[0, i-1]$ từ cha của nó và giá trị $[i, j-1]$ từ con trái của nó, nên nó đơn giản tính toán $[0, j-1] \leftarrow [0, i-1] \otimes [i, k]$ và gửi giá trị này cho con phải.

Hình 29.9 nêu mạch kết quả, bao gồm trường hợp cận nảy sinh tại gốc. Giá trị $x_0 = [0, 0]$ được cung cấp dưới dạng đầu vào tại gốc, và một thành phần ⊗ nữa được dùng để tính toán (nói chung) giá trị $y_n = [0, n] = [0, 0] \otimes [1, n]$.

Nếu n là một lũy thừa chính xác của 2, thì mạch tiền tố song song sử dụng $2n - 1$ thành phần. Nó chỉ mất $O(\lg n)$ thời gian để tính toán tất cả $n + 1$ tiền tố, bởi phép tính tiếp tục tiến lên cây rồi trở xuống. Bài tập 29.2-5 nghiên cứu độ sâu của mạch chi tiết hơn.

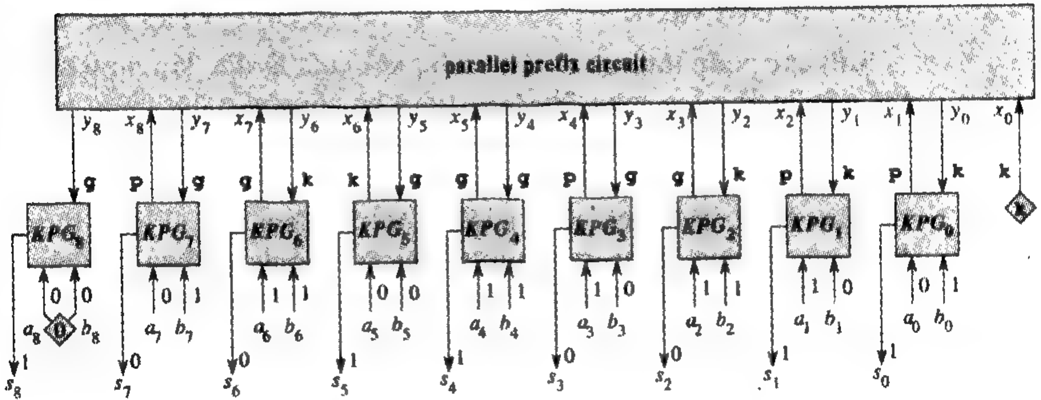


Hình 29.9 Một mạch tiền tố song song với $n = 8$. **(a)** Cấu trúc chung của mạch. và các giá trị được nhớ trên mỗi dây dẫn. **(b)** Cùng một mạch có các giá trị tương ứng với các Hình 29.3 và 29.7.

Hoàn tất bộ cộng nhớ kiểm tra trước

Giờ đây, sau khi có một mạch tiền tố song song, ta có thể hoàn thành phần mô tả của bộ cộng nhớ kiểm tra trước. Hình 29.10 nêu phần kiến tạo. Một ***bộ cộng nhớ kiểm tra trước*** n-bit bao gồm $n + 1$ hộp ***KPG***, mỗi hộp có kích cỡ $\Theta(1)$, và một mạch tiền tố song song có các đầu vào x_0, x_1 ,..., x_n (x_0 được đấu chết với k) và các kết xuất y_0, y_1,..., y_n. Hộp KPG ***KPG_i*** nhận các đầu vào bên ngoài a_i và b_i và tạo ra bit tổng s_i. (Các bit đầu vào a_n và b_n được đấu chết với 0.) Cho a_{i-1}, và b_{i-1}, hộp KPG_{i-1} tính toán $x_i \in$ {**k,p,g**} theo phương trình (29.3) và gửi giá trị này dưới dạng đầu vào bên ngoài x_i của mạch tiền tố song song. (Giá trị của x_{n+1} được bỏ qua.) Việc tính toán toàn bộ x_i mất $\Theta(1)$ thời gian. Sau một độ trễ $O(\lg n)$, mạch tiền tố song song tạo ra y_0, y_1 ,..., y_n. Theo Bổ đề 29.1, y_i là **k** hoặc g; nó không thể là **p**. Mỗi giá trị y_i nêu rõ trường hợp nhớ-trong cho bộ toàn cộng FA_i trong bộ cộng nhớ lược: y_i= **k** hàm ý $c_i = 0$, và y_i = **g** hàm ý $c_i = 1$. Như vậy, giá trị của y_i được nạp vào KPG_i để nêu rõ c_i

nhớ-trong và bit tổng s_i = parity(a_i, b_i, c_i) được tạo trong thời gian bất biến. Như vậy, bộ cộng nhớ kiểm tra trước hoạt động trong $O(\lg n)$ thời gian và có kích cỡ $\Theta(n)$.



Hình 29.10 Phần kiến tạo của một bộ cộng nhớ kiểm tra trước n-bit, được nêu ở đây với n = 8. Nó bao gồm n + 1 hộp KPG KPG_i với i = 0, 1,..., n. Mỗi hộp KPG_i nhận các đầu vào bên ngoài a_i và b_i (ở đó a_n và b_n được đấu chết với 0, như được nêu bằng hình thoi) và tính toán tình trạng nhớ x_{i+1}. Các giá trị này được nạp vào mạch tiền tố song song, nó trả về các kết quả y_i của phép tính tiền tố. Mỗi hộp KPG_i giờ đây nhận y_i làm đầu vào, diễn dịch nó dưới dạng bit nhớ-trong c_i, rồi kết xuất bit tổng s_i = parity(a_i, b_i, c_i). Ở đây cũng nêu các giá trị mẫu tương ứng với các giá trị nêu trong các Hình 29.3 và 29.9.

29.2.3 Phép cộng nhớ tiết kiệm

Một bộ cộng nhớ kiểm tra trước có thể cộng hai số n-bit trong $O(\lg n)$ thời gian. Thật đáng ngạc nhiên, việc cộng ba số n-bit chỉ mất thêm một thời lượng không đổi. Điểm tinh tế đó là rút gọn bài toán cộng ba số thành bài toán cộng chỉ hai số.

Cho ba số n -bit $x = \langle x_{n-1}, x_{n-2}, \ldots, x_0\rangle$, $y = \langle y_{n-1}, y_{n-2}, \ldots, y_0\rangle$, và $z = \langle z_{n-1}, z_{n-2}, \ldots, z_0\rangle$, một bộ cộng nhớ tiết kiệm [carry-save addition] n-bit tạo ra một số n-bit $u = \langle u_{n-1}, u_{n-2}, \ldots, u_0\rangle$ và một số (n + 1) bit $v = \langle v_n, v_{n-1}, \ldots, v_0\rangle$ sao cho

$$u + v = x + y + z .$$

Như đã nêu trong Hình 29.11(a), nó thực hiện điều này bằng cách tính toán

$$u_i = \text{parity}(x_i, y_i, z_i) ,$$

$$v_{i+1} = \text{majority}(x_i, y_i, z_i) ,$$

với i = 0, 1,..., n − 1. Bit v_0 luôn bằng 0.

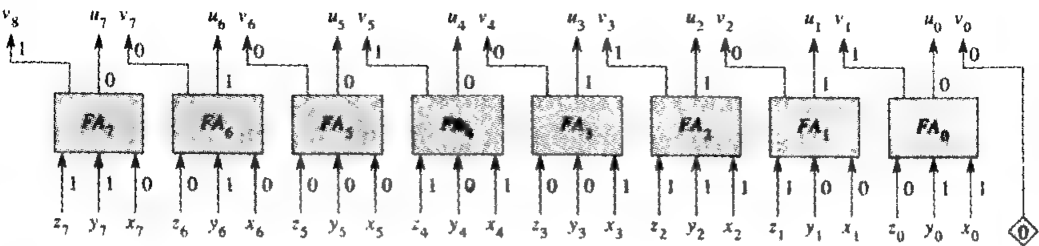
Bộ cộng nhớ tiết kiệm n-bit nêu trong Hình 29.11(b) bao gồm n bộ

toàn cộng FA_0, FA_1,..., FA_{n-1}. Với $i = 0, 1,..., n - 1$, bộ toàn cộng FA_i nhận các đầu vào x_i, y_i, và z_i. Kết xuất bít tổng của FA_i được nhận dưới dạng u_i, và trường hợp nhớ-ngoài của FA_i được nhận dưới dạng v_{i+1}, . Bit v_0 được đấu chết với 0.

Bởi các phép tính tất cả $2n + 1$ bit kết xuất là độc lập, nên chúng có thể tiến hành song song. Như vậy, một bộ cộng nhớ tiết kiệm vận hành trong $\Theta(1)$ thời gian và có kích cỡ $\Theta(n)$. Do đó, để tính tổng ba số n-bit, ta chỉ cần thực hiện một phép cộng nhớ tiết kiệm, mất $\Theta(1)$ thời gian, rồi thực hiện một phép cộng nhớ kiểm tra trước, mất $O(\lg n)$ thời gian. Mặc dù phương pháp này theo tiệm cận không tốt hơn phương pháp dùng hai phép cộng nhớ kiểm tra trước, song trong thực tế nó nhanh hơn nhiều. Hơn nữa, như sẽ thấy trong Đoạn 29.3, phép cộng nhớ tiết kiệm là trung tâm đối với các thuật toán nhanh dành cho phép nhân.

8	7	6	5	4	3	2	1	0		i
	0	0	0	1	1	1	0	1	=	x
	1	1	0	0	0	1	0	1	=	y
	1	0	0	1	0	1	1	0	=	z
	0	1	0	0	1	1	1	0	=	u
1	0	0	1	0	1	0	1	0	=	v

(a)



(b)

Hình 29.11 (a) Phép cộng nhớ tiết kiệm. Cho ba số n-bit x, y, và z, ta tạo một số n-bit u' và một số $(n+1)$ bit v sao cho $x + y + z = u + v$. Cặp bit tô bóng thứ i là một hàm của x_i, y_i, và z_i. **(b)** Một bộ cộng nhớ tiết kiệm 8-bit. Mỗi bộ toàn cộng FA_i nhận các đầu vào x_i, y_i, và z_i và tạo ra bit tổng u_i và bit nhớ-ngoài v_{i+1}. Bit v_0 được đấu chết với 0.

Bài tập

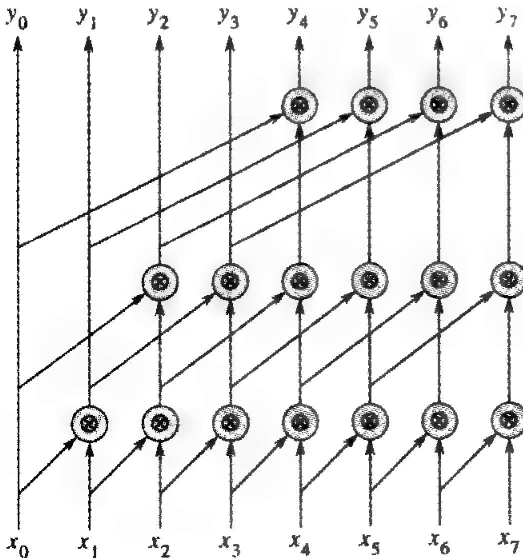
29.2-1

Cho $a = <01111111>$, $b = <00000001>$, và $n = 8$. Nêu bit tổng và bit nhớ được các bộ toàn cộng kết xuất khi phép cộng nhớ lược được thực

hiện trên hai dãy này. Nêu các tình trạng nhớ $x_0, x_1, ..., x_8$ tương ứng với a và b, gắn nhãn mỗi dây dẫn của mạch tiền tố song song của Hình 29.9 bằng giá trị mà nó đã cung cấp cho các đầu vào x_i này, và nêu các kết xuất kết quả $y_0, y_1, ..., y_8$.

29.2-2

Chứng minh toán tử tình trạng nhớ ⊗ căn cứ vào Hình 29.5 mang tính kết hợp.



Hình 29.12 Một mạch tiền tố song song để dùng trong Bài tập 29.2-6.

29.2-3

Nêu bằng ví dụ cách kiến tạo một mạch tiền tố song song $O(\lg n)$-thời gian với các giá trị của n không phải là các lũy thừa chính xác của 2 bằng cách vẽ một mạch tiền tố song song với $n = 11$. Định rõ đặc điểm khả năng thực hiện của các mạch tiền tố song song xây dựng theo hàng dạng các cây nhị phân tùy ý.

29.2-4

Nêu phần kiến tạo cấp cổng của hộp KPG_i. Giả sử mỗi kết xuất x_i được biểu thị bởi <00> nếu x_i = **k**, bởi <11> nếu x_i = **g**, và bởi <01> hoặc <10> nếu x_i = **p**. Cũng giả sử mỗi đầu vào y_i được biểu thị bởi 0 nếu y_i = **k** và bởi 1 nếu y_i = **g**.

29.2-5

Gán nhãn cho mỗi dây dẫn trong mạch tiền tố song song của Hình 29.9(a) bằng độ sâu của nó. Một ***lộ trình tới hạn*** trong một mạch là một

lộ trình có số lượng thành phần tổ hợp lớn nhất trên một lộ trình bất kỳ từ các đầu vào đến các kết xuất. Định danh lộ trình tới hạn trong Hình 29.9(a), và chứng tỏ chiều dài của nó là $O(\lg n)$. Chứng tỏ một nút nào đó có các thành phần ⊗ hoạt động trong $\Theta(\lg n)$ thời gian tách rời. Có thể có một nút mà các thành phần ⊗ của nó hoạt động đồng thời không?

29.2-6

Nêu một sơ đồ khối đệ quy của mạch trong Hình 29.12 với bất kỳ số n đầu vào nào là một lũy thừa chính xác của 2. Dựa vào sơ đồ khối của bạn, chứng minh mạch quả thực có thực hiện một phép tính tiền tố. Chứng tỏ độ sâu của mạch là $\Theta(\lg n)$ và nó có kích cỡ $\Theta(n \lg n)$.

29.2-7

Nêu mức lùa ra cực đại của bất kỳ dây dẫn nào trong bộ cộng nhớ kiểm tra trước? Chứng tỏ phép cộng vẫn có thể được thực hiện trong $O(\lg n)$ thời gian bởi một mạch có kích cỡ $\Theta(n)$ cho dù ta hạn chế các cổng theo mức lùa ra $O(1)$.

29.2-8

Một ***mạch kiểm*** [tally circuit] có n đầu vào nhị phân và $m = \lceil \lg(n+1) \rceil$ kết xuất. Được diễn dịch dưới dạng một số nhị phân, các kết xuất cho số lượng 1 trong các đầu vào. Ví dụ, nếu đầu vào là <10011110>, kết xuất là <101>, nêu rõ có năm số 1 trong đầu vào. Mô tả một mạch kiểm có độ sâu $O(\lg n)$ với kích cỡ $\Theta(n)$.

29.2-9 *

Chứng tỏ có thể hoàn tất phép cộng n-bit với một mạch tổ hợp có độ sâu 4 và kích cỡ đa thức trong n nếu các cổng AND và OR được phép mức lùa vào cao tùy ý. (*Tùy chọn*: Đạt độ sâu 3.)

29.2-10 *

Giả sử hai số n-bit ngẫu nhiên được cộng với một bộ cộng nhớ lược, ở đó mỗi bit là 0 hoặc 1 một cách độc lập với xác suất bằng nhau. Chứng tỏ với xác suất ít nhất $1 - 1/n$, không có trường hợp nhớ lan truyền xa hơn các giai đoạn liên tiếp $O(\lg n)$. Nói cách khác, mặc dù độ sâu của bộ cộng nhớ lược là $\Theta(n)$, với hai số ngẫu nhiên, các kết xuất hầu như luôn ổn định trong $O(\lg n)$ thời gian.

29.3 Các mạch nhân

Thuật toán phép nhân "tiểu học" trong Hình 29.13 có thể tính toán tích 2n-bit $p = \langle p_{2n-1}, p_{2n-2}, \ldots, p_0 \rangle$ của hai số n-bit $a = \langle a_{n-1}, a_{n-2}, \ldots, a_0 \rangle$ và $b = \langle b_{n-1}, b_{n-2}, \ldots, b_0 \rangle$. Ta xét các bit của b, từ b_0 lên đến b_{n-1}. Với mỗi bit b_i có một giá trị của 1, ta cộng a vào tích, nhưng được chuyển trái i vị trí. Với mỗi bit b_i có một giá trị của 0, ta cộng vào 0. Như vậy, cho $m^{(i)} = a \bullet b_i \bullet 2^i$, ta tính toán

$$p = a \bullet b = \sum_{i=0}^{n-1} m^{(i)} .$$

Mỗi số hạng $m^{(i)}$ được gọi là một ***tích riêng phần*** [partial product]. Có n tích riêng phần để tính tổng, với các bit tại các vị trí 0 đến $2n - 2$. Trường hợp nhớ-ngoài từ bit cao nhất cho ra bit cuối tại vị trí $2n - 1$.

Trong đoạn này, ta xét hai mạch để nhân hai số n-bit. Các bộ nhân mảng vận hành trong $\Theta(n)$ thời gian và có kích cỡ $\Theta(n^2)$. Các bộ nhân cây Wallace cũng có kích cỡ $\Theta(n^2)$, nhưng chúng vận hành trong $\Theta(\lg n)$ thời gian. Cả hai mạch đều dựa trên thuật toán tiểu học.

				1	1	1	0	=	a
				1	1	0	1	=	b
				1	1	1	0	=	$m^{(0)}$
			0	0	0	0		=	$m^{(1)}$
		1	1	1	0			=	$m^{(2)}$
	1	1	1	0				=	$m^{(3)}$
1	0	1	1	0	1	1	0	=	p

Hình 29.13 Phương pháp nhân "tiểu học", được nêu ở đây nhân $a = \langle 1110 \rangle$ với $b = \langle 1101 \rangle$ để có được tích $p = \langle 10110110 \rangle$. Ta cộng $\sum_{i=0}^{n-1} m^{(i)}$, ở đó $m^{(i)} = a \bullet b_i \bullet 2^i$. Ở đây, $n = 8$. Mỗi số hạng $m^{(i)}$ được hình thành bằng cách chuyển a (nếu $b_i = 1$) hoặc 0 (nếu $b_i = 0$) i vị trí về bên trái. Các bit không được nêu là 0 bất chấp các giá trị của a và b.

29.3.1 Các bộ nhân mảng

Về khái niệm, một bộ nhân mảng bao gồm ba phần. Phần đầu tiên tạo thành các tích riêng phần. Phần thứ hai tính tổng các tích riêng phần dùng các bộ cộng nhớ tiết kiệm. Cuối cùng, phần thứ ba tính tổng hai con số xuất phát từ các phép cộng nhớ tiết kiệm dùng một bộ cộng nhớ lược hoặc nhớ kiểm tra trước.

Hình 29.14 nêu một ***bộ nhân mảng*** với hai số nhập $a = <a_{n-1}, a_{n-2}, ..., a_0>$ và $b = <b_{n-1}, b_{n-2}, ..., b_0>$. Các giá trị a_i chạy theo chiều dọc, và các giá trị b_i chạy theo chiều ngang. Mỗi bit đầu vào lùa ra [fans out] đến n cổng AND để tạo thành các tích riêng phần. Các bộ toàn cộng, được tổ chức dưới dạng các bộ cộng nhớ tiết kiệm, tính tổng các tích riêng phần. Các bit cấp thấp của tích cuối là kết xuất bên phải. Các bit có thứ tự cao hơn được lập thành bằng cách cộng hai số kết xuất bởi bộ cộng nhớ tiết kiệm.

Ta hãy xét phần kiến tạo của bộ nhân mảng kỹ hơn. Cho hai số nhập $a = <a_{n-1}, a_{n-2}, ..., a_0>$ và $b = <b_{n-1}, b_{n-2}, ..., b_0>$, các bit của các tích riêng phần thường dễ tính toán. Cụ thể, với i, $j = 0, 1, ..., n-1$, ta có

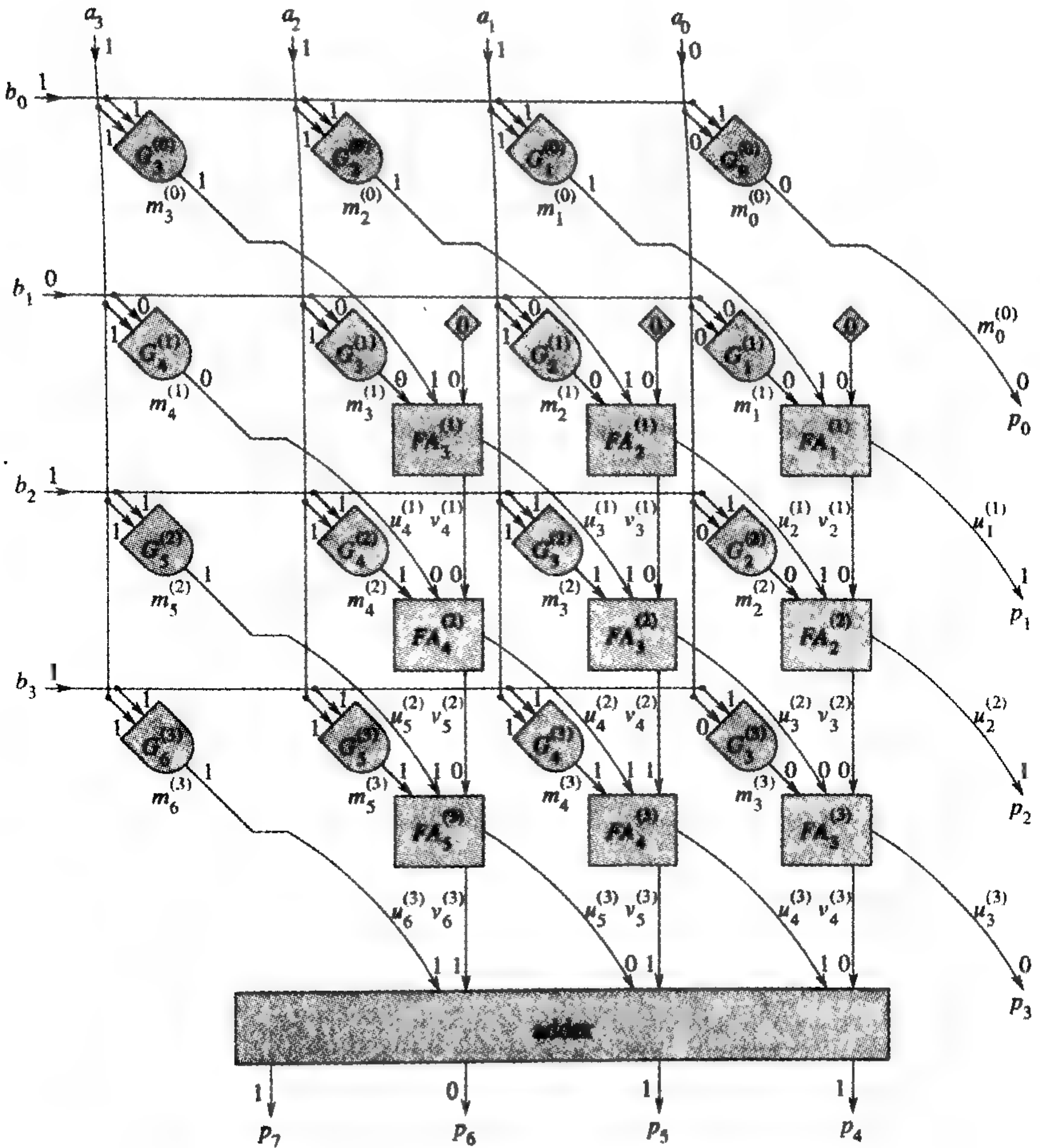
$$m^{(i)}_{j+1} = a_j \bullet b_j .$$

Do có thể tính toán trực tiếp tích các giá trị 1-bit với một cổng AND, nên tất cả các bit của các tích riêng phần (ngoại trừ những bit được xem là 0, không cần tính toán rõ ràng) có thể được tạo trong một bước bằng n^2 cổng AND.

Hình 29.15 minh họa cách thức mà bộ nhân mảng thực hiện các phép cộng nhớ tiết kiệm khi tính tổng các tích riêng phần trong Hình 29.13. Nó bắt đầu bằng cách cộng nhớ tiết kiệm $m^{(0)}$, $m^{(1)}$, và 0, cho ra một số $(n+1)$ bit $u^{(1)}$ và một số $(n+1)$ bit $v^{(1)}$. (Số $v^{(1)}$ chỉ có $n+1$ bit, không có $n+2$, bởi các bit thứ $(n+1)$ của cả 0 lẫn $m^{(0)}$ là 0.) Như vậy, $m^{(0)} + m^{(1)} = u^{(1)} + v^{(1)}$. Sau đó, nó cộng nhớ tiết kiệm $u^{(1)}$, $v^{(1)}$, và $m^{(2)}$ cho ra một số $(n+2)$ bit $u^{(2)}$ và một số $(n+2)$ bit $v^{(2)}$. (Một lần nữa, $v^{(2)}$ chỉ có $n+2$ bit bởi cả $u^{(1)}_{n+2}$ lẫn $v^{(1)}_{n+2}$ là 0.) Như vậy, ta có $m^{(0)} + m^{(1)} + m^{(2)} = u^{(2)} + v^{(2)}$. Bộ nhân tiếp tục, cộng nhớ tiết kiệm $u^{(i-1)}$, $v^{(i-1)}$, và $m^{(i)}$ với $i = 2, 3, ..., n-1$. Kết quả là một số $(2n-1)$ bit $u^{(n-1)}$ và một số $(2n-1)$ bit $v^{(n-1)}$, ở đó

$$\begin{aligned} u^{(n-1)} + v^{(n-1)} &= \sum_{i=0}^{n-1} m^{(i)} \\ &= p . \end{aligned}$$

Thực vậy, các phép cộng nhớ tiết kiệm trong Hình 29.15 hoạt động trên số lượng bit nhiều hơn mức thật sự cần thiết. Nhận thấy với $i = 1, 2, ..., n-1$ và $j = 0, 1, ..., i-1$, ta có $m^{(i)}_j = 0$ do cách ta chuyển các tích riêng phần. Cũng nhận thấy $v^{(i)}_j = 0$ với $i = 1, 2, ..., n-1$ và $j = 0, 1, ..., i,$



Hình 29.14 Một bộ nhân mảng tính toán tích $p = \langle p_{2n-1}, p_{2n-2}, ..., p_0 \rangle$ của hai số n-bit $a = \langle a_{n-1}, a_{n-2}, ..., a_0 \rangle$ và $b = \langle b_{n-1}, b_{n-2}, ..., b_0 \rangle$, được nêu ở đây với $n = 4$. Mỗi cổng AND $G_j^{(i)}$ tính toán bit tích riêng phần $m_j^{(i)}$. Mỗi hàng các bộ toàn cộng hình thành một bộ cộng nhớ tiết kiệm. n bit thấp của tích là $m_0^{(0)}$ và các bit u xuất phát từ cột mút phải của các bộ toàn cộng. n bit tích cao được hình thành bằng cách cộng các bit u *và* v xuất phát từ hàng đáy các bộ toàn cộng. Được nêu là các giá trị bit cho các đầu vào $a = \langle 1110 \rangle$ và $b = \langle 1101 \rangle$ và tích $p = \langle 10110110 \rangle$, tương ứng với các Hình 29.13 và 29.15.

				0	0	0	0	=	0
				1	1	1	0	=	$m^{(0)}$
			0	0	0	0		=	$m^{(1)}$
			0	1	1	1	0	=	$u^{(1)}$
			0	0	0			=	$v^{(1)}$
		1	1	1	0			=	$m^{(2)}$
		1	1	0	1	1	0	=	$u^{(2)}$
		0	1	0				=	$v^{(2)}$
	1	1	1	0				=	$m^{(3)}$
	1	0	1	0	1	1	0	=	$u^{(3)}$
	1	1	0					=	$v^{(3)}$
1	0	1	1	0	1	1	0	=	p

Hình 29.15 Đánh giá tổng các tích riêng phần bằng phép cộng nhớ tiết kiệm được lặp lại. Với ví dụ này, a = <1110> và b = <1101>. Các bit trắng là 0 bất chấp các giá trị của a và b. Trước tiên ta đánh giá $m^{(0)} + m^{(1)} + 0 = u^{(1)} + v^{(1)}$, thì $u^{(1)} + v^{(1)} + m^{(2)} = u^{(2)} + v^{(2)}$, sau đó $u^{(2)} + v^{(2)} + m^{(3)} = u^{(3)} + v^{(3)}$, và cuối cùng $p = m^{(0)} + m^{(1)} + m^{(2)} + m^{(3)} = u^{(3)} + v^{(3)}$. Lưu ý, $p_0 = m_0^{(0)}$ và $p_i = u_i^{(i)}$ với i = 1, 2,..., $n - 1$.

$i + n$, $i + n + 1$,..., $2n - 1$. (Xem Bài tập 29.3-1.) Do đó, mỗi phép cộng nhớ tiết kiệm cần vận hành trên chỉ $n - 1$ bit.

Giờ đây, ta xét sự tương ứng giữa bộ nhân mảng và lược đồ cộng nhớ tiết kiệm được lặp lại. Mỗi cổng AND được gán nhãn bằng $G^{(i)}$ với một i và j trong các miền $0 \le i \le n - 1$ và $0 \le j \le 2n - 2$. Cổng $G^{(i)}$ tạo ra $m_j^{(i)}$, bit thứ j của tích riêng phần thứ i. Với i = 0, 1,..., $n - 1$, hàng thứ i của các cổng AND tính toán n bit quan trọng của tích riêng phần $m^{(i)}$, nghĩa là, $\langle m^i_{n+i-1}, m^i_{n+i-2}, ..., m^i_i \rangle$.

Ngoại trừ các bộ toàn cộng trong hàng đỉnh (nghĩa là, với i = 2, 3,..., $n - 1$), mỗi bộ toàn cộng $FA_j^{(i)}$ nhận ba bit đầu vào—$m_j^{(i)}$, $u_j^{(i-1)}$, và $v^{(i-1)}$—và tạo ra hai bit kết xuất—$u_j^{(i)}$ và $v_{j+1}^{(i)}$. (Lưu ý, trong cột mút trái của các bộ toàn cộng, $u_{i+n-1}^{(i-1)} = m_{i+n-1}^{(i)}$.) Mỗi bộ toàn cộng $FA_j^{(1)}$ trong hàng đỉnh nhận các đầu vào $m^{(0)}$, $m^{(1)}$, và 0 rồi tạo các bit $u_j^{(1)}$ và $v_{j+1}^{(1)}$.

Cuối cùng, ta hãy xét kết xuất của bộ nhân mảng. Như đã nhận xét trên đây, $v_j^{(n-1)} = 0$ với j = 0, 1,..., $n - 1$. Như vậy, $p_j = u_j^{(n-1)}$ với j = 0, 1,..., $n - 1$. Hơn nữa, bởi $m^{(1)} = 0$, ta có $u^{(1)} = m^{(0)}$, và bởi i bit có thứ tự thấp nhất của mỗi $m^{(i)}$ và $v^{(i-1)}$ là 0, ta có $u_j^{(i)} = u_j^{(i-1)}$ với i = 2, 3,..., $n - 1$ và j = 0, 1,..., i -1. Như vậy, $p_0 = m^{(0)}$ và, theo quy nạp, $p_i = u^{(i)}$ với i = 1, 2,...,

$n - 1$. Các bit tích $\langle p_{2n-1}, p_{2n-2}, \ldots, p_n \rangle$ được tạo bởi một bộ cộng n-bit cộng các kết xuất từ hàng cuối các bộ toàn cộng:

$$\langle p_{2n-1}, p_{2n-2}, \ldots, p_n \rangle =$$
$$\langle u^{(n-1)}_{2n-2}, u^{(n-1)}_{2n-3}, \ldots, u^{(n-1)}_{n} \rangle + \langle v^{(n-1)}_{2n-2}, v^{(n-1)}_{2n-3}, \ldots, v^{(n-1)}_{n} \rangle .$$

Phân tích

Dữ liệu lược qua một bộ nhân mảng từ trên trái đến dưới phải. Nó mất $\Theta(n)$ thời gian để tạo các bit tích cấp thấp $\langle p_{n-1}, p_{n-2}, \ldots, p_0 \rangle$, và mất $\Theta(n)$ thời gian để tạo sẵn sàng các đầu vào đến bộ cộng. Nếu bộ cộng là một bộ cộng nhớ lược, nó mất một $\Theta(n)$ thời gian khác để các bít tích thứ tự cao $\langle p_{2n-1}, p_{2n-2}, \ldots, p_n \rangle$ nổi lên. Nếu bộ cộng là một bộ cộng nhớ kiểm tra trước, ta chỉ cần $\Theta(\lg n)$ thời gian, nhưng tổng thời gian vẫn là $\Theta(n)$.

Có n^2 cổng AND và $n^2 - n$ bộ toàn cộng trong bộ nhân mảng. Bộ cộng cho các bit kết xuất thứ tự cao chỉ đóng góp một $\Theta(n)$ cổng khác. Như vậy, bộ nhân mảng có kích cỡ $\Theta(n^2)$.

29.3.2 Bộ nhân cây Wallace

Một ***cây Wallace*** là một mạch rút gọn bài toán lấy tổng n số n-bit thành bài toán lấy tổng hai số $\Theta(n)$ bit. Để thực hiện, nó dùng $\lfloor n/3 \rfloor$ bộ cộng nhớ tiết kiệm song song để chuyển đổi tổng của n số thành tổng của $\lceil 2n/3 \rceil$ số. Sau đó, nó xây dựng đệ quy một cây Wallace trên $\lceil 2n/3 \rceil$ số kết quả. Theo cách này, tập hợp các con số được rút gọn dần cho đến khi chỉ còn hai số. Nhờ thực hiện nhiều phép cộng nhớ tiết kiệm song song, các cây Wallace cho phép nhân hai số n-bit trong $0(\lg n)$ thời gian dùng một mạch có kích cỡ $\Theta(n^2)$.

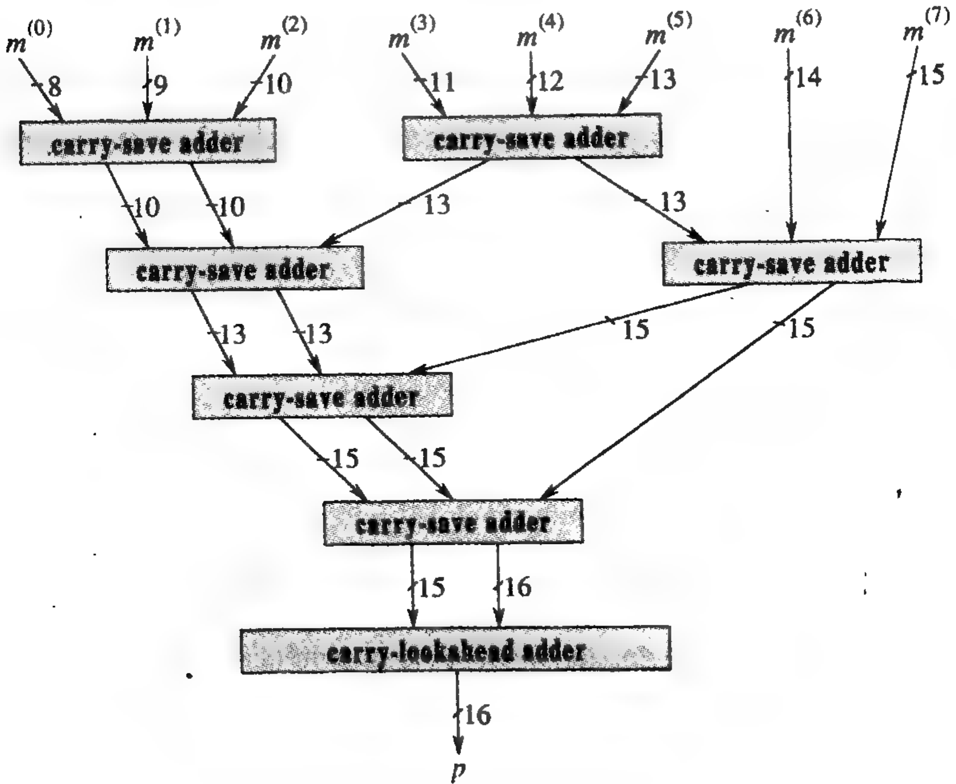
Hình 29.16 nêu một cây Wallace[2] cộng 8 tích riêng phần $m^{(0)}, m^{(1)}, \ldots, m^{(7)}$. Tích riêng phần $m^{(i)}$ bao gồm $n + i$ bit. Mỗi dòng biểu diễn một số nguyên, chứ không chỉ một bit đơn lẻ; cạnh mỗi dòng là số bit mà dòng biểu diễn (xem Bài tập 29.3-3). Bộ cộng nhớ kiểm tra trước ở đáy sẽ cộng một số $(2n - 1)$ bit vào một số $2n$-bit để cho ra tích $2n$ bit.

---

[2] Như đã thấy trong hình, một cây Wallace không phải thực sự là cây, nhưng thay vì thế là một đồ thị phi chu trình có hướng. Tên là theo lịch sử.

Phân tích

Thời gian mà một cây Wallace n-đầu vào yêu cầu tùy thuộc vào độ sâu của các bộ cộng nhớ tiết kiệm. Tại mỗi cấp của cây, mỗi nhóm 3 số được rút gọn còn 2 số, với tối đa 2 số chừa lại (như trong trường hợp của $m^{(6)}$ và $m^{(7)}$ tại cấp trên cùng). Như vậy, độ sâu cực đại $D(n)$ của một bộ cộng nhớ tiết kiệm trong một cây Wallace n-đầu vào sẽ căn cứ vào phép truy toán



Hình 29.16 Một cây Wallace cộng $n = 8$ tích riêng phần $m^{(0)}, m^{(1)},..., m^{(7)}$. Mỗi dòng biểu diễn một số với số lượng bit được nêu rõ. Kết xuất bên trái của mỗi bộ cộng nhớ tiết kiệm biểu diễn các bit tổng, và kết xuất bên phải biểu diễn các bit nhớ.

$$D(n)= \begin{cases} 0 & \text{nếu } n \leq 2\,, \\ 1 & \text{nếu } n = 3\,, \\ D(\lceil 2n/3 \rceil) + 1 & \text{nếu } n \geq 4\,, \end{cases}$$

có giải pháp $D(n) = \Theta(\lg n)$ theo trường hợp 2 của định lý chủ (Định lý 4.1). Mỗi bộ cộng nhớ tiết kiệm mất $\Theta(1)$ thời gian. Tất cả n tích

riêng phần có thể được lập thành trong $\Theta(1)$ thời gian song song. ($i - 1$ bit thứ tự thấp của $m^{(i)}$, với $i = 1, 2, ..., n - 1$, được đấu chết với 0.) Bộ cộng nhớ kiểm tra trước chiếm $O(\lg n)$ thời gian. Như vậy, toàn bộ phép nhân hai số n-bit mất $\Theta(\lg n)$ thời gian.

Một bộ nhân cây Wallace với hai số n-bit có kích cỡ $\Theta(n^2)$, mà ta có thể thấy như sau. Trước tiên ta định cận kích cỡ mạch của các bộ cộng nhớ tiết kiệm. Có thể dễ dàng đạt được một cận dưới $\Omega(n^2)$, bởi có $\lfloor 2n/3 \rfloor$ bộ cộng nhớ tiết kiệm tại độ sâu 1, và mỗi bộ bao gồm ít nhất n bộ toàn cộng. Để có cận trên $O(n^2)$, ta nhận thấy do tích cuối có $2n$ bit, nên mỗi bộ cộng nhớ tiết kiệm trong cây Wallace chứa tối đa $2n$ bộ toàn cộng. Ta cần chứng tỏ có tất thảy $O(n)$ bộ cộng nhớ tiết kiệm. Cho $C(n)$ là tổng các bộ cộng nhớ tiết kiệm trong một cây Wallace với n con số nhập. Ta có phép truy toán

$$C(n) \le \begin{cases} 1 & \text{nếu } n = 3 , \\ C(\lceil 2n/3 \rceil) + \lfloor n/3 \rfloor & \text{nếu } n \ge 4 , \end{cases}$$

có giải pháp $C(n) = \Theta(n)$ theo trường hợp 3 của định lý chủ. Như vậy ta được một cận sát tiệm cận có kích cỡ $\Theta(n^2)$ cho các bộ cộng nhớ tiết kiệm của một bộ nhân cây Wallace. Hệ mạch để xác lập n tích riêng phần có kích cỡ $\Theta(n^2)$, và bộ cộng nhớ kiểm tra trước ở cuối có kích cỡ $\Theta(n)$. Như vậy, kích cỡ của toàn bộ bộ nhân là $\Theta(n^2)$.

Mặc dù theo tiệm cận bộ nhân dựa trên cây Wallace nhanh hơn bộ nhân mảng và có cùng kích cỡ tiệm cận, song bố cục củanó khi được thực thi sẽ không bình thường như bộ nhân mảng, cũng không "trù mật" bằng (theo nghĩa có ít không gian lãng phí giữa các thành phần mạch). Trong thực tế, ta thường dùng một kiểu thỏa hiệp giữa hai thiết kế. Ý tưởng đó là sử dụng hai mảng song song, một mảng cộng dồn nửa của các tích riêng phần và một mảng cộng dồn nửa kia. Độ trễ lan truyền chỉ là phân nửa so với mức gánh chịu của một mảng cộng dồn đơn lẻ tất cả n tích riêng phần. Thêm hai phép cộng nhớ tiết kiệm sẽ rút gọn 4 số mà các mảng kết xuất còn 2 số, và sau đó một bộ cộng nhớ kiểm tra trước sẽ cộng 2 số để cho ra tích. Tổng độ trễ lan truyền hơi nhiều hơn một nửa so với một bộ nhân mảng đầy đủ, cộng với một kỳ hạn $O(\lg n)$ bổ sung.

Bài tập

29.3-1

Chứng minh trong một bộ nhân mảng, $v_j^{(i)} = 0$ với $i = 1, 2, \ldots, n - 1$ và $j = 0, 1, \ldots, i, i + n, i + n + 1, \ldots, 2n - 1$.

29.3-2

Chứng tỏ trong bộ nhân mảng của Hình 29.14, ngoại trừ một còn tất cả các bộ toàn cộng trong hàng đỉnh đều không cần thiết. Bạn cần tiến hành đấu dây lại.

29.3-3

Giả sử một bộ cộng nhớ tiết kiệm nhận các đầu vào x, y, và z rồi tạo ra các kết xuất s và c, với n_x, n_y, n_z, n_s, và n_c bit theo thứ tự nêu trên. Không để mất tính tổng quát, ta cũng giả sử $n_x \le n_y \le n_z$. Chứng tỏ $n_s = n_z$ và s

$$n_c = \begin{cases} n_z & \text{nếu } n_y < n_z, \\ n_z + 1 & \text{nếu } n_y = n_z. \end{cases}$$

29.3-4

Chứng tỏ vẫn có thể thực hiện phép nhân trong $O(\lg n)$ thời gian với kích cỡ $O(n^2)$ cho dù ta hạn chế các cổng để có mức lùa ra $O(1)$.

29.3-5

• Mô tả một mạch hiệu quả để tính toán số thương khi chia một số nhị phân x cho 3. (*Mách nước*: Lưu ý, trong nhị phân, $.010101\ldots = .01 \times 1.01 \times 1.0001 \times \ldots$.)

29.3-6

Một ***bộ chuyển chu trình*** [cyclic shifter], hoặc ***bộ chuyển baren*** [barrel shifter], là một mạch có hai đầu vào $x = \langle x_{n-1}, x_{n-2}, \ldots, x_0 \rangle$ và $s = \langle s_{m-1}, s_{m-2}, \ldots, s_0 \rangle$, ở đó $m = \lceil \lg n \rceil$. Kết xuất của nó $y = \langle y_{n-1}, y_{n-2}, \ldots, y_0 \rangle$ được chỉ định bởi $y_i = x_{i+s \bmod n}$, với $i = 0, 1, \ldots, n - 1$. Nghĩa là, bộ chuyển quay các bit của x theo lượng đãđịnh bởi s. Mô tả một bộ chuyển chu trình hiệu quả. Theo dạng phép nhân modula, một bộ chuyển chu trình sẽ thực thi chức năng nào?

29.4 Các mạch gắn đồng hồ

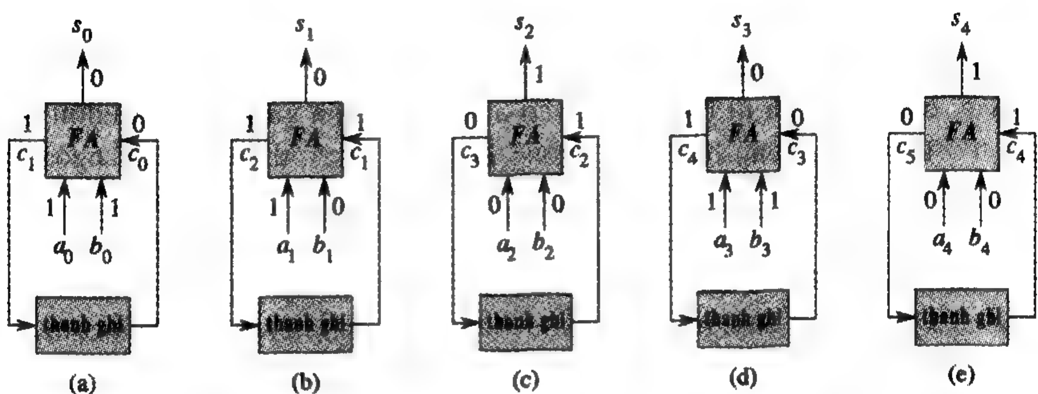
Các thành phần của một mạch tổ hợp chỉ được dùng một lần trong

một phép tính. Nhờ đưa các thành phần bộ nhớ gắn đồng hồ vào mạch, ta có thể dùng lại các thành phần tổ hợp. Bởi chúng có thể dùng phần cứng nhiều lần, nên các mạch gắn đồng hồ thường có thể nhỏ hơn nhiều so với các mạch tổ hợp với cùng chức năng.

Đoạn này nghiên cứu các mạch gắn đồng hồ để thực hiện phép cộng và phép nhân. Ta bắt đầu bằng một mạch gắn đồng hồ có kích cỡ $\Theta(1)$, có tên là bộ cộng bit nối tiếp, có thể cộng hai số n-bit trong $\Theta(n)$ thời gian. Sau đó, ta nghiên cứu các bộ nhân mảng tuyến tính. Ta trình bày một bộ nhân mảng tuyến tính có kích cỡ $\Theta(n)$ có thể nhân hai số n-bit trong $\Theta(n)$ thời gian.

29.4.1 Phép cộng bit nối tiếp

Để giới thiệu khái niệm của một mạch gắn đồng hồ, ta trở về bài toán cộng hai số n-bit. Hình 29.17 nêu cách dùng một bộ toàn cộng đơn lẻ để tạo ra tổng $(n + 1)$ bit $s = \langle s_n, s_{n-1}, ..., s_0 \rangle$ của hai số n-bit $a = \langle a_{n-1}, a_{n-2}, ..., a_0 \rangle$ và $b = \langle b_{n-1}, b_{n-2}, ..., b_0 \rangle$. Thế giới bên ngoài trình bày các bit đầu vào mỗi lần một cặp: đầu tiên a_0 và b_0, sau đó a_1 và b_1, vân vân. Mặc dù ta muốn trường hợp nhớ-ngoài từ một vị trí bit là nhớ-trong cho vị trí bit kế tiếp, song ta không thể chỉ nạp thẳng kết xuất c của bộ toàn cộng vào một đầu vào. Có một vấn đề tính giờ: c_i nhớ-trong nhập vào bộ toàn cộng phải tương ứng với các đầu vào thích hợp a_i và b_i. Trừ phi các bit đầu vào này đến chính xác cùng một lúc với nhớ phản hồi, kết xuất có thể không đúng.



Hình 29.17 Phép toán của một bộ cộng bit nối tiếp. Trong kỳ đồng hồ thứ i, với $i = 0, 1, ..., n$, bộ toàn cộng FA nhận các bit đầu vào a_i và b_i từ thế giới bên ngoài và một bit nhớ c_i từ thanh ghi. Sau đó, bộ toàn cộng kết xuất bit tổng s_i, được cung cấp theo bên ngoài, và bit nhớ c_{i+1} được lưu trữ trở lại trong thanh ghi để dùng trong kỳ đồng hồ kế tiếp. Thanh ghi được khởi tạo với $c_0 = 0$. **(a)-(e)** Trạng thái của mạch của mỗi trong số năm kỳ đồng hồ trong phép cộng của $a = \langle 1011 \rangle$ và $b = \langle 1001 \rangle$ để tạo ra $s = \langle 10100 \rangle$.

Như Hình 29.17 đã nêu, giải pháp đó là sử dụng một ***mạch gắn đồng hồ*** [clocked circuit], hoặc ***mạch tuần tự***, bao gồm hệ mạch tổ hợp và một hoặc nhiều ***thanh ghi*** (các thành phần bộ nhớ gắn đồng hồ). Hệ mạch tổ hợp có các đầu vào từ thế giới bên ngoài hoặc từ kết xuất của các thanh ghi. Nó cung cấp các kết xuất cho thế giới bên ngoài và cho đầu vào của các thanh ghi. Như trong các mạch tổ hợp, ta ngăn cấm hệ mạch tổ hợp trong một mạch gắn đồng hồ chứa các chu trình.

Mỗi thanh ghi trong một mạch gắn đồng hồ được điều khiển bởi một tín hiệu định kỳ, hoặc ***đồng hồ***. Mỗi khi đồng hồ tạo xung, hoặc ***đánh nhịp*** [tick], thanh ghi nạp vào và lưu trữ giá trị tại đầu vào của nó. Thời gian giữa các nhịp đồng hồ liên tiếp là một ***kỳ đồng hồ*** [clock period]. Trong một ***mạch gắn đồng hồ toàn cục***, mọi thanh ghi làm việc theo cùng một đồng hồ.

Ta hãy xét chi tiết hơn phép toán của một thanh ghi. Ta xem mỗi nhịp đồng hồ như một xung tạm thời. Tại một nhịp đã cho, một thanh ghi đọc giá trị đầu vào được trình bày cho nó *vào lúc đó* rồi lưu trữ nó. Sau đó, giá trị được lưu trữ xuất hiện tại kết xuất của thanh ghi, ở đó nó có thể được dùng để tính toán các giá trị được dời vào các thanh ghi khác tại nhịp đồng hồ kế tiếp. Nói cách khác, giá trị tại đầu vào của một thanh ghi trong một kỳ đồng hồ sẽ xuất hiện trên kết xuất của thanh ghi trong kỳ đồng hồ kế tiếp.

Giờ đây, hãy xét mạch trong Hình 29.17, mà ta gọi là ***bộ cộng bit nối tiếp***. Để các kết xuất của bộ toàn cộng được đúng đắn, ta yêu cầu kỳ đồng hồ ít nhất phải dài bằng độ trễ lan truyền của bộ toàn cộng, sao cho hệ mạch tổ hợp có một cơ hội để ổn định giữa các nhịp đồng hồ. Trong kỳ đồng hồ 0, nêu trong Hình 29.17(a), thế giới bên ngoài áp dụng các bit đầu vào a_0 và b_0 cho hai trong số các đầu vào của bộ toàn cộng. Ta mặc nhận rằng thanh ghi được khởi tạo để lưu trữ một 0; như vậy bit nhớ-trong ban đầu, là kết xuất thanh ghi, là $c_0 = 0$. Về sau trong kỳ đồng hồ này, bit tổng s_0 và c_1 nhớ-ngoài nổi lên từ bộ toàn cộng. Bit tổng đi đến thế giới bên ngoài, ở đó có lẽ nó sẽ được lưu dưới dạng một phần của nguyên cả tổng s. Dây dẫn từ phần nhớ-ngoài của b_0 toàn cộng sẽ nạp vào thanh ghi, sao cho c_1 được đọc vào thanh ghi theo nhịp đồng hồ kế tiếp. Do đó, tại đầu kỳ đồng hồ 1, thanh ghi chứa c_1. Trong kỳ đồng hồ 1, xem trong Hình 29.17(b), thế giới bên ngoài áp dụng a_1 và b_1 cho bộ toàn cộng, mà khi đọc c_1 từ thanh ghi, nó tạo ra các kết xuất s_1 và c_2. Bit tổng s_1 đi ra thế giới bên ngoài, và c_2 đi đến thanh ghi.

Chu trình này tiếp tục cho đến kỳ đồng hồ n, nêu trong Hình 29.17(e). ở đó thanh ghi chứa c_n. Sau đó, thế giới bên ngoài áp dụng $a_n = b_n = 0$, sao cho ta có $s_n = c_n$.

Phân tích

Để xác định tổng thời gian t được tiếp nhận bởi một mạch gắn đồng hồ toàn cục, ta cần biết số p các kỳ đồng hồ và thời hạn d của mỗi kỳ đồng hồ: $t = pd$. Kỳ đồng hồ d phải đủ dài để toàn bộ hệ mạch tổ hợp ổn định giữa các nhịp. Mặc dù với vài đầu vào nó có thể ổn định sớm hơn, nếu mạch làm việc đúng đắn với tất cả các đầu vào, d phải ít nhất tỷ lệ với độ sâu của hệ mạch tổ hợp.

Ta hãy xem phải mất bao lâu để cộng hai số n-bit theo bit-nối tiếp. Mỗi kỳ đồng hồ mất $\Theta(1)$ thời gian bởi độ sâu của bộ toàn cộng là $\Theta(1)$. Do cần có $n + 1$ nhịp đồng hồ để tạo ra tất cả các kết xuất, nên tổng thời gian để thực hiện phép cộng bit nối tiếp là $(n + 1)\,\Theta(1) = \Theta(n)$.

Kích cỡ của bộ cộng bit nối tiếp (số lượng các thành phần tổ hợp cộng với số lượng các thanh ghi) là $\Theta(1)$.

Phép cộng nhớ lược đối lại phép cộng bit nối tiếp

Nhận thấy một bộ cộng nhớ lược cũng giống như một bộ cộng bit nối tiếp sao lặp có các thanh ghi được thay bằng các tuyến nối trực tiếp giữa các thành phần tổ hợp. Nghĩa là, bộ cộng nhớ lược tương ứng với một "tiến trình trải rộng" [unrolling] không gian trong phép tính của bộ cộng bit nối tiếp. Bộ toàn cộng thứ i trong bộ cộng nhớ lược thực thi kỳ đồng hồ thứ i của bộ cộng bit nối tiếp. Nói chung, ta có thể thay một mạch gắn đồng hồ bằng một mạch tổ hợp tương đương có cùng độ trễ thời gian tiệm cận nếu ta biết trước số lượng kỳ đồng hồ mà mạch gắn đồng hồ sẽ chạy. Tất nhiên, cũng có sự trả giá. Mạch gắn đồng hồ sử dụng các thành phần mạch ít hơn (ít hơn một thừa số $\Theta(n)$ cho bộ cộng bit nối tiếp đối lại bộ cộng nhớ lược), nhưng mạch tổ hợp có ưu điểm là ít hệ mạch điều khiển hơn—ta không cần đồng hồ hoặc mạch đồng bộ hóa bên ngoài để trình bày các bit đầu vào và lưu trữ các bit tổng. Hơn nữa, mặc dù các mạch có cùng độ trễ thời gian tiệm cận, song trong thực tế mạch tổ hợp thường chạy hơi nhanh hơn. Tốc độ phụ trội là khả dĩ bởi mạch tổ hợp không phải đợi các giá trị để ổn định trong mỗi kỳ đồng hồ. Nếu tất cả các đầu vào ổn định ngay, các giá trị chỉ việc lược qua mạch theo tốc độ khả dĩ cực đại, mà không đợi đồng hồ.

a	b
19	29
9	58
4	116
2	232
1	464
	551

(a)

a	b
10011	11101
1001	111010
100	1110100
10	11101000
1	111010000
	1000100111

(b)

Hình 29.18 Nhân 19 với 29 với thuật toán nông dân Nga. Khoản nhập cột *a* trong mỗi hàng là phân nửa khoản nhập của hàng trước đó với các phân số được bỏ qua, và các khoản nhập cột *b* sẽ nhân đôi từ hàng này qua hàng khác. Ta cộng các khoản nhập cột *b* trong tất cả các hàng với các khoản nhập cột *a* lẻ, được tô bóng. Tổng này là tích mong muốn. **(a)** Các con số được diễn tả bằng thập phân. **(b)** Cùng các con số nhưng bằng nhị phân.

29.4.2 Các bộ nhân mảng tuyến tính

Các bộ nhân tổ hợp trong Đoạn 29.3 cần kích cỡ $\Theta(n^2)$ để nhân hai số n-bit. Giờ đây, ta trình bày hai bộ nhân tuyến tính, thay vì hai chiều, các mảng các thành phần mạch. Cũng như bộ nhân mảng, trong hai bộ nhân mảng tuyến tính này bộ nhanh hơn sẽ chạy trong $\Theta(n)$ thời gian.

Các bộ nhân mảng tuyến tính thực thi ***thuật toán nông dân Nga*** (sở dĩ gọi như vậy là vì bởi các cư dân phương Tây viếng thăm Nga trong thế kỷ thứ mười chín đã thấy thuật toán này được dùng rộng rãi ở đây), xem Hình 29.18(a). Cho hai con số nhập *a* và *b*, ta tạo hai cột số, đứng đầu là *a* và *b*. Trong mỗi hàng, khoản nhập cột *a* là phân nửa khoản nhập cột *a* của hàng trước đó, với các phân số được thải bỏ. Khoản nhập cột *b* gấp hai lần khoản nhập cột *b* của hàng trước đó . Hàng cuối là hàng có một khoản nhập cột *a* là 1. Ta xét tất cả các khoản nhập cột *a* chứa các giá trị lẻ và tổng các khoản nhập cột *b* tương ứng. Tổng này là tích $a \cdot b$.

Mặc dù thoạt nhìn thuật toán nông dân Nga có vẻ như đáng nể, Hình 29.18(b) chứng tỏ thực tế nó chỉ là một thực thi hệ thống số nhị phân của phương pháp nhân tiểu học, nhưng với các con số được diễn tả theo thập phân thay vì nhị phân. Các hàng ở đó khoản nhập cột *a* là lẻ sẽ đóng góp cho tích một số hạng *b* được nhân với lũy thừa thích hợp của 2.

Một thực thi mảng tuyến tính chậm

Hình 29.19(a) nêu một cách để thực thi thuật toán nông dân Nga với hai số n-bit. Ta dùng một mạch gắn đồng hồ bao gồm một mảng tuyến tính $2n$ ô. Mỗi ô chứa ba thanh ghi. Một thanh ghi lưu giữ một bit từ một khoản nhập a, một thanh ghi lưu giữ một bit từ một khoản nhập b, và một thanh ghi lưu giữ một bit của tích p. Ta dùng các con chữ trên để thể hiện các giá trị ô trước mỗi bước của thuật toán. Ví dụ, giá trị của bit a_i trước bước thứ j là $a_i^{(j)}$, và ta định nghĩa $a_i^{(j)} = \langle a_{2n-1}^{(j)}, a_{2n-2}^{(j)}, \ldots, a_0^{(j)} \rangle$.

Thuật toán thi hành một dãy n bước, được đánh số $0, 1, \ldots, n - 1$, ở đó mỗi bước chiếm một kỳ đồng hồ. Thuật toán duy trì sự bất biến rằng trước bước thứ j,

$$a^{(j)} \bullet b^{(j)} + p^{(j)} = a \bullet b \qquad (29.6)$$

(xem Bài tập 29.4-2). Thoạt đầu, $a^{(0)} = a$, $b^{(0)} = b$, và $p^{(0)} = 0$. Bước thứ j bao gồm các phép tính sau đây.

1. Nếu $a^{(j)}$ là lẻ (nghĩa là, $a_0^{(j)} = 1$), thì cộng b vào p: $p^{(j+1)} \leftarrow b^{(j)} + p^{(j)}$.

(Phép cộng được thực hiện bằng một bộ cộng nhớ lược chạy theo chiều dài của mảng; các bit nhớ lược từ phải qua trái.) Nếu $a^{(j)}$ là chẵn, thì nhớ p qua tới bước kế tiếp: $p^{(j+1)} \leftarrow p^{(j)}$.

2. Chuyển a sang phải một vị trí bit:

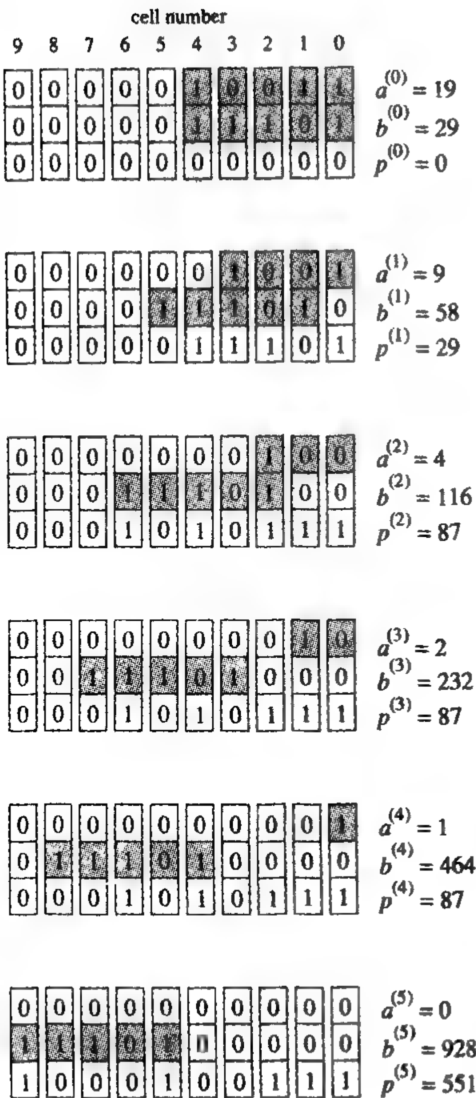
$$a_{(i)}^{(j+1)} \leftarrow \begin{cases} a_{i+1}^{(j)} & \text{nếu } 0 \le i \le 2n-2, \\ 0 & \text{nếu } i = 2n-1. \end{cases}$$

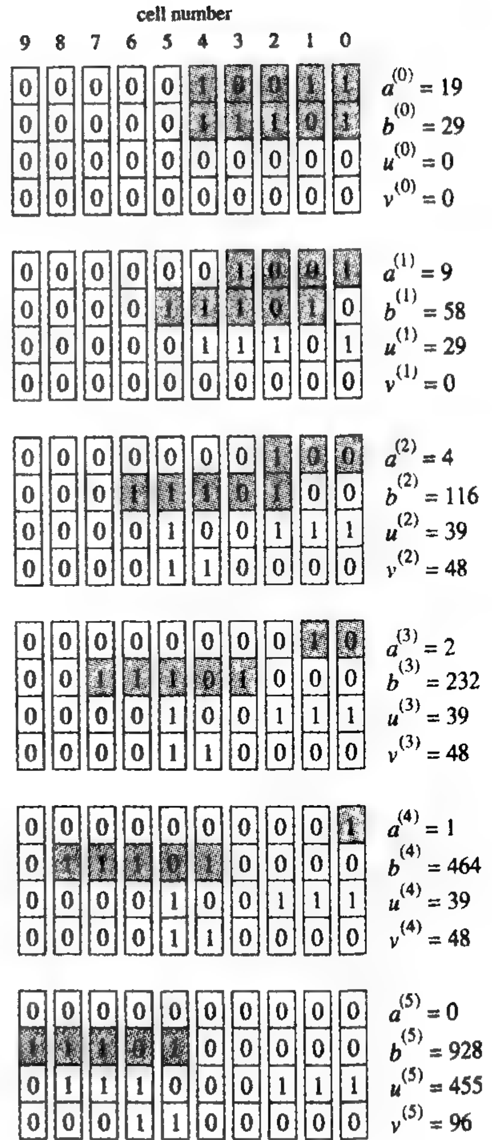
3. Chuyển b sang trái một vị trí bit:

$$b_{(i)}^{(j+1)} \leftarrow \begin{cases} b_{i-1}^{(j)} & \text{nếu } 1 \le i \le 2n-1, \\ 0 & \text{nếu } i = 0. \end{cases}$$

Sau khi chạy n bước, ta đã chuyển ra tất cả các bit của a; như vậy, $a^{(n)} = 0$. Như vậy bất biến (29.6) hàm ý rằng $p^{(n)} = a \bullet b$.

Giờ đây ta phân tích thuật toán. Có n bước, giả định thông tin điều khiển đồng thời truyền bá đến từng ô. Mỗi bước mất $\Theta(n)$ thời gian trong trường hợp xấu nhất, bởi độ sâu của bộ cộng nhớ lược là $\Theta(n)$, và như vậy thời hạn của kỳ đồng hồ ít nhất phải là $\Theta(n)$. Mỗi bước chuyển chỉ mất $\Theta(1)$ thời gian. Do đó, nói chung thuật toán chiếm $\Theta(n^2)$ thời gian. Bởi mỗi ô có kích cỡ không đổi, nên nguyên cả mảng tuyến tính có kích cỡ $\Theta(n)$.





Hình 29.19 Hai kiểu thực thi mảng tuyến tính của thuật toán nông dân Nga, nêu phép nhân của $a = 19 = \langle 10011 \rangle$ cho $b = 29 = \langle 11101 \rangle$, với $n = 5$. Tình huống tại đầu mỗi bước j được nêu, với các bit quan trọng còn lại của $a^{(j)}$ và $b^{(j)}$ được tô bóng. **(a)** Một thực thi chậm chạy trong $\Theta(n^2)$ thời gian. Bởi $a^{(5)} = 0$, nên ta có $p^{(5)} = a \bullet b$. Có n bước, và mỗi bước sử dụng một phép cộng nhớ lược. Do đó, kỳ đồng hồ tỷ lệ với chiều dài của mảng, hoặc $\Theta(n)$, dẫn đến $\Theta(n^2)$ thời gian nói chung. **(b)** Một thực thi nhanh chạy trong $\Theta(n)$ thời gian bởi mỗi bước sử dụng phép cộng nhớ tiết kiệm thay vì phép cộng nhớ lược, như vậy chỉ mất $\Theta(1)$ thời gian. Có một tổng $2n - 1 = 9$ bước; sau bước cuối đã nêu, phép cộng nhớ tiết kiệm lặp lại của u và v cho ra $u^{(9)} = a \bullet b$.

Một thực thi mảng tuyến tính nhanh

Nhờ dùng phép cộng nhớ tiết kiệm thay vì cộng nhớ lược, ta có thể giảm thời gian cho mỗi bước là Θ(1), nhờ đó cải thiện thời gian chung là Θ(n). Như Hình 29.19(b) đã nêu, một lần nữa mỗi ô chứa một bit của một khoản nhập a và một bit của một khoản nhập b. Mỗi ô cũng chứa thêm hai bit, từ u và v, là các kết xuất từ phép cộng nhớ tiết kiệm. Dùng một phần biểu diễn nhớ tiết kiệm để tích lũy tích, ta duy trì sự bất biến rằng trước bước thứ j,

$$a^{(j)} \bullet b^{(j)} + u^{(j)} + v^{(j)} = a \bullet b \qquad (29.7)$$

(một lần nữa, xem Bài tập 29.4-2). Mỗi bước sẽ chuyển a và b theo cùng cách như kiểu thực thi chậm, sao cho ta có thể tổ hợp các phương trình (29.6) và (29.7) để cho ra $u^{(j)} + v^{(j)} = p^{(j)}$. Như vậy, các bit u và v chứa cùng thông tin như các bit p trong kiểu thực thi chậm.

Bước thứ j của kiểu thực thi nhanh thực hiện phép cộng nhớ tiết kiệm trên u và v, ở đó các toán hạng tùy thuộc vào việc u là lẻ hay chẵn. Nếu $a_0^{(j)} = 1$, ta tính toán

$$u_i^{(j+1)} \leftarrow \text{parity}(b_i^j, u_i^j, v_i^j) \text{ với } i = 0, 1, ..., 2n-1$$

và

$$v_i^{(j+1)} \leftarrow \begin{cases} \text{majority}(b_{i-1}^j, u_{i-1}^j, v_{i-1}^j & \text{nếu } 1 \le i \le 2n-1, \\ 0 & \text{nếu } i = 0. \end{cases}$$

Bằng không, $a_0 = 0$, và ta tính toán

$$u_i^{(j+1)} \leftarrow \text{parity}(0, u_i^j, v_i^j) \text{ với } i = 0, 1, ..., 2n-1$$

và

$$v_i^{(j+1)} \leftarrow \begin{cases} \text{majority}(0, u_{i-1}^j, v_{i-1}^j) & \text{nếu } 1 \le i \le 2n-1, \\ 0 & \text{nếu } i = 0. \end{cases}$$

Sau khi cập nhật u và v, bước thứ j sẽ chuyển a sang phải và b sang trái giống như kiểu thực thi chậm.

Kiểu thực thi chậm thực hiện một tổng $2n$ - 1 bước. Với $j \ge n$, ta có $a^{(j)} = 0$, và do đó bất biến (29.7) ý rằng $u^{(j)} + v^{(j)} = a \bullet b$. Một khi $a^{(j)} = 0$, tất cả các bước tiếp nữa đều chỉ được dùng để cộng nhớ tiết kiệm u và v. Bài tập 29.4-3 yêu cầu bạn chứng tỏ $v^{(2n-1)} = 0$, sao cho $u^{(2n-1)} = a \bullet b$.

Tổng thời gian trong trường hợp xấu nhất là Θ(n), bởi mỗi trong số $2n$ - 1 bước sẽ mất Θ(1) thời gian. Bởi mỗi ô vẫn có kích cỡ không đổi, tổng kích cỡ giữ nguyên là Θ(n).

Bài tập

29.4-1

Cho a = <101101>, b = <011110>, và $n = 6$. Nêu cách hoạt động của thuật toán nông dân Nga, trong cả thập phân lẫn nhị phân, với các đầu vào a và b.

29.4-2

Chứng minh các bất biến (29.6) và (29.7) cho các bộ nhân mảng tuyến tính.

29.4-3

Chứng minh trong bộ nhân mảng tuyến tính nhanh, $v^{(2n-1)} = 0$.

29.4-4

Mô tả cách thức mà bộ nhân mảng trong Đoạn 29.3.1 biểu diễn một tiến trình "trải rộng" [unrolling] trong phép tính của bộ nhân mảng tuyến tính nhanh.

29.4-5

Xét một luồng dữ liệu $<x_1, x_2, ...>$ đi đến một mạch gắn đồng hồ với tốc độ 1 giá trị cho mỗi nhịp đồng hồ. Với một giá trị cố định n, mạch phải tính toán giá trị

$$y_t = \max_{t-n+1 \le i \le t} x_i$$

với $t = n, n + 1, ...$Nghĩa là, y_t là cực đại của n giá trị vừa mới được mạch tiếp nhận. Nêu một mạch có kích cỡ $O(n)$ mà trên mỗi nhịp đồng hồ sẽ nhập giá trị x_t và tính toán giá trị kết xuất y_t trong $O(1)$ thời gian. Mạch có thể dùng các thanh ghi và các thành phần tổ hợp tính toán cực đại của hai đầu vào.

29.4-6 *

Làm lại Bài tập 29.4-5 chỉ dùng $O(\lg n)$ thành phần "cực đại".

Các Bài Toán

29-1 Các mạch chia

Ta có thể kiến tạo một mạch chia từ các mạch trừ và nhân bằng một kỹ thuật có tên ***sự lặp lại Newton.*** Ta sẽ tập trung vào bài toán tính toán một số nghịch đảo hữu quan, bởi ta có thể có một mạch chia bằng cách

thực hiện một phép nhân bổ sung.

Ý tưởng đó là tính toán một dãy y_0, y_1, y_2,...của các số xấp xỉ với với số nghịch đảo của một số x bằng công thức

$y_{i+1} \leftarrow 2y_i - xy_i^2$.

Giả sử x được gán dưới dạng một phân số nhị phân n-bit trong miền $1/2 \le x \le 1$. Bởi số nghịch đảo có thể là một phân số lặp lại vô hạn, ta sẽ tập trung vào việc tính toán một số xấp xỉ n-bit chính xác đến bit ít quan trọng nhất của nó.

a. Giả sử rằng $|y_i - 1/x| \le \varepsilon$ với một hằng $\varepsilon > 0$. Chứng minh $|y_{i+1} - 1/x| \le \varepsilon^2$.

b. Nêu một số xấp xỉ ban đầu y_0 sao cho y_k thỏa $|y_k - 1/x| \le 2 - 2^k$ với tất cả $k \ge 0$. k phải lớn tới mức nào để số xấp xỉ y_k chính xác tới bit ít quan trọng nhất của nó?

c. Mô tả một mạch tổ hợp mà, căn cứ vào một đầu vào n-bit x, nó tính toán một số xấp xỉ n-bit với $1/x$ trong $O(\lg^2 n)$ thời gian. Nêu kích cỡ của mạch? (*Mách nước:* Với một ít thông minh, bạn có thể đánh bại cận kích cỡ của $\Theta(n^2 \lg n)$.)

29-2 Các công thức bool cho các hàm đối xứng

Một hàm n-đầu vào $f(x_1, x_2,..., x_n)$ là ***đối xứng*** nếu

$f(x_1, x_2,..., x_n) = f(x_{\pi(1)}, x_{\pi(2)},..., x_{\pi(n)})$

với bất kỳ phép hoán vị π của $\{1, 2,..., n\}$. Trong bài toán này, ta chứng tỏ có một công thức bool biểu thị cho f có kích cỡ đa thức trong n. (Với mục tiêu của chúng ta, một công thức bool là một chuỗi bao gồm các biến $x_1, x_2,..., x_n$, các dấu ngoặc đơn, và các toán tử bool $\vee$, $\wedge$, và $\neg$) Cách tiếp cận của chúng ta đó là chuyển đổi một mạch bool có độ sâu loga thành một công thức bool có kích cỡ đa thức tương đương. Ta mặc nhận tất cả các mạch được kiến tạo từ các cổng AND 2-đầu vào, OR 2-đầu vào, và NOT.

a. Để bắt đầu, ta xét một hàm đối xứng đơn giản. ***Hàm phần lớn*** được tổng quát hóa trên n đầu vào bool được định nghĩa bởi

$$\text{majority}_n(x_1, x_2,..., x_n) = \begin{cases} 1 & \text{nếu } x_1 + x_2 + ... + x_n > n/2 , \\ 0 & \text{bằng không} . \end{cases}$$

Mô tả một mạch tổ hợp có độ sâu $O(\lg n)$ với majority$_n$. (*Mách nước:* Xây dựng một cây các bộ cộng.)

b. Giả sử f là một hàm bool tùy ý gồm n biến bool $x_1, x_2,..., x_n$. Giả sử thêm rằng có một mạch C có độ sâu d tính toán f. Nêu cách kiến tạo từ C một công thức bool với f có chiều dài $O(2^d)$. Kết luận rằng có công thức kích cỡ đa thức cho majority$_n$.

c. Chứng tỏ mọi hàm bool đối xứng $f(x_1, x_2,..., x_n)$ đều có thể được diễn tả dưới dạng một hàm của $\Sigma_i x_i$.

d. Chứng tỏ mọi hàm đối xứng trên n đầu vào bool có thể được tính toán bằng một mạch tổ hợp có độ sâu $O(\lg n)$.

e. Chứng tỏ mọi hàm bool đối xứng trên n biến bool có thể được biểu diễn bởi một công thức bool có chiều dài đa thức trong n.

Ghi chú Chương

Hầu hết các sách về số học máy tính đều tập trung nhiều về các kiểu thực thi thực tiễn của hệ mạch hơn là lý thuyết thuật toán. Savage [173] là một trong số ít người nghiên cứu các khía cạnh thuật toán của chủ đề. Các sách thiên nhiều về phần cứng nói về số học máy tính của Cavanagh [39] và Hwang [108] là những cuốn đặc biệt nên đọc. Các cuốn sách hay về thiết kế logic tuần tự và tổ hợp bao gồm Hill và Peterson [96]; và Kohavi [126], với phần xoáy vào lý thuyết ngôn ngữ hình thức.

Aiken và Hopper [7] phác họa lịch sử tiên khởi của các thuật toán số học. Phép cộng nhớ lược ít nhất cũng xưa bằng bàn tính tàu, vào khoảng trên 5000 năm. Bộ tính cơ học đầu tiên sử dụng phép cộng nhớ lược đã được B. Pascal sáng chế vào năm 1642. Một máy tính toán thích ứng phép cộng lặp lại cho phép nhân đã được diễn đạt bởi S. Morland vào năm 1666 và độc lập bởi G. W. Leibnitz vào năm 1671. Thuật toán nông dân Nga dùng cho phép nhân dường như cũ hơn nhiều so với việc dùng nó ở Nga vào thế kỷ thứ mười chín. Theo Knuth [122], nó được dùng bởi các nhà toán học Ai cập cách đây 1800 năm trước công nguyên.

Các tình trạng triệt, phát sinh, và lan truyền của một xích nhớ đã được khai thác trong bộ rờ le [relay calculator] xây dựng tại Harvard vào giữa những năm 1940 [180]. Một trong những thực thi đầu tiên của phép cộng nhớ kiểm tra trước đã được Weinberger và Smith [199] mô tả, nhưng phương pháp kiểm tra trước của họ yêu cầu các cổng lớn. Ofman [152] đã chứng minh có thể cộng các số n-bit trong $O(\lg n)$ thời gian dùng phép cộng nhớ kiểm tra trước với các cổng có kích cỡ không đổi.

Ý tưởng dùng phép cộng nhớ tiết kiệm để tăng tốc phép nhân là

của Estrin, Gilchrist, và Pomerene [64]. Atrubin [13] mô tả một bộ nhân mảng tuyến tính có chiều dài vô hạn có thể dùng để nhân các số nhị phân có chiều dài tùy ý. Bộ nhân tạo ra bit thứ *n* của tích ngay khi tiếp nhận các bit thứ *n* của các đầu vào. Bộ nhân cây Wallace được xem là của Wallace [197], nhưng ý tưởng cũng đã được khám phá độc lập bởi Ofman [152].

Các thuật toán chia có từ thời I. Newton, là người mà vào khoảng 1665 đã phát minh cái được xem là phép lặp lại Newton. Bài toán 29-1 sử dụng phép lặp lại Newton để kiến tạo một mạch chia có độ sâu $\Theta(\lg^2 n)$. Phương pháp này đã được cải thiện bởi Beame, Cook, và Hoover [19], họ đã chứng tỏ phép chia *n*-bit thực tế có thể thực hiện trong độ sâu $\Theta(\lg n)$.

30 Các Thuật Toán Cho Các Máy Tính Song Song

Khi các máy tính xử lý song song sinh sối nảy nở, lợi ích cũng đã gia tăng trong ***các thuật toán song song:*** các thuật toán thực hiện nhiều phép toán cùng một lúc. Cuộc nghiên cứu các thuật toán song song giờ đây đã phát triển thành một lĩnh vực nghiên cứu riêng của nó. Quả vậy, các thuật toán song song đã được phát triển cho nhiều bài toán mà ta đã giải trong sách này bằng các thuật toán nối tiếp bình thường. Chương này sẽ mô tả một vài thuật toán song song đơn giản minh họa các vấn đề và các kỹ thuật căn bản.

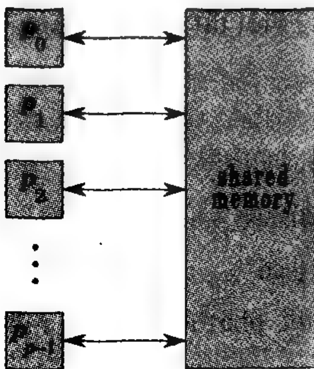
Để nghiên cứu các thuật toán song song, ta phải chọn một mô hình thích hợp để tính toán song song. Tất nhiên, máy truy cập ngẫu nhiên, hoặc RAM, mà ta đã dùng xuyên suốt cuốn sách này là nối tiếp thay vì song song. Các mô hình song song mà ta đã nghiên cứu—các mạng sắp xếp (Chương 28) và các mạch (Chương 29)—tỏ ra quá hạn chế để, chẳng hạn, thẩm tra các thuật toán trên các cấu trúc dữ liệu.

Các thuật toán song song trong chương này được trình bày theo dạng một mô hình lý thuyết phổ dụng: máy truy cập ngẫu nhiên song song [parallel random-access machine], hoặc PRAM (đọc là "pi ram"). Ta có thể dễ dàng mô tả nhiều thuật toán song song cho các mảng, các danh sách, các cây, và đồ thị theo mô hình PRAM. Mặc dù PRAM bỏ qua nhiều khía cạnh quan trọng của máy song song thực, song các thuộc tính chủ chốt của các thuật toán song song có khuynh hướng vượt trên các mô hình mà chúng được thiết kế. Nếu một thuật toán PRAM làm tốt hơn một thuật toán PRAM khác, thì về cơ bản khả năng thực hiện tương đối ắt không thay đổi khi cả hai thuật toán được thích ứng để chạy trên một máy tính song song thực.

Mô hình PRAM

Hình 30.1 nêu kiến trúc cơ bản của ***máy truy cập ngẫu nhiên song song (PRAM)***. Có p bộ xử lý (nối tiếp) bình thường $P_0, P_1, \ldots, P_{p-1}$ có một bộ nhớ toàn cục, dùng chung làm kho lưu trữ. Tất cả các bộ xử lý có thể đọc hoặc ghi từ bộ nhớ toàn cục theo chế độ "song song" (cùng một lúc). Các bộ xử lý cũng có thể thực hiện nhiều phép toán số học và lôgic theo chế độ song song.

Giả thiết chính liên quan đến khả năng thực hiện thuật toán trong mô hình PRAM đó là thời gian thực hiện có thể được đo dưới dạng số lượng các lần truy cập bộ nhớ song song mà một thuật toán thực hiện. Giả thiết này là một phép tổng quát hóa đơn giản của mô hình RAM bình thường, ở đó theo tiệm cận số lần truy cập bộ nhớ cũng tốt như mọi kiểu đo khác của thời gian thực hiện. Giả thiết đơn giản này sẽ giúp ta rất nhiều trong khi nghiên cứu các thuật toán song song, cho dù các máy tính song song thực không thể thực hiện các lần truy cập song song vào bộ nhớ toàn cục trong thời gian đơn vị: thời gian mà một lần truy cập bộ nhớ tăng trưởng cùng với số lượng bộ xử lý trong máy tính song song.



Hình 30.1 Kiến trúc cơ bản của PRAM. Có p bộ xử lý $P_0, P_1, \ldots, P_{p-1}$ nối với một bộ nhớ dùng chung. Mỗi bộ xử lý có thể truy cập một từ tùy ý của bộ nhớ dùng chung trong thời gian đơn vị.

Tuy vậy, với các thuật toán song song truy cập dữ liệu theo cách tùy ý, giả thiết về các phép toán bộ nhớ thời gian đơn vị có thể được chứng minh. Các máy song song thực thường có một mạng truyền thông có thể hỗ trợ sự trừu tượng của một bộ nhớ toàn cục. Việc truy cập dữ liệu thông qua mạng là một phép toán tương đối chậm so sánh với phép toán số học và các phép toán khác. Như vậy, việc đếm số lần truy cập bộ nhớ song song mà hai thuật toán song song thi hành, thực tế, cho ra một ước lượng khá chính xác về các khả năng thực hiện tương đối của chúng. Cách chính qua đó các máy thực vi phạm sự trừu tượng thời gian đơn vị của PRAM đó là có vài khuôn mẫu truy cập bộ nhớ tỏ ra nhanh hơn những khuôn mẫu khác. Tuy nhiên, với tư cách là một phép xấp xỉ đầu tiên, giả thiết thời gian đơn vị trong mô hình PRAM tỏ ra khá hợp lý.

Thời gian thực hiện của một thuật toán song song tùy thuộc vào số lượng bộ xử lý thi hành thuật toán cũng như kích cỡ của bài toán đầu vào. Do đó, nói chung, ta phải đề cập cả số đếm bộ xử lý lẫn thời gian khi phân tích các thuật toán PRAM; điều này trái ngược với các thuật

toán nối tiếp, mà khi phân tích ta chỉ tập trung chủ yếu vào thời gian. Thông thường, có sự trả giá giữa số lượng bộ xử lý mà một thuật toán sử dụng và thời gian thực hiện củanó. Đoạn 30.3 mô tả các trường hợp trả giá này.

Truy cập bộ nhớ đồng thời và loại trừ

Thuật toán ***đọc đồng thời*** là một thuật toán PRAM mà trong khi thi hành nhiều bộ xử lý có thể đồng thời đọc từ cùng vị trí bộ nhớ dùng chung. Thuật toán ***đọc loại trừ*** là một thuật toán PRAM ở đó không có hai bộ xử lý đồng thời đọc cùng vị trí bộ nhớ. Ta tiến hành phân biệt như vậy đối với việc nhiều bộ xử lý có thể đồng thời ghi vào cùng vị trí bộ nhớ hay không, chia các thuật toán PRAM thành các thuật toán ***ghi đồng thời*** và ***ghi loại trừ***. Dưới đây là dạng viết tắt thường dùng cho các kiểu thuật toán mà ta gặp

- ***EREW***: đọc loại trừ và ghi loại trừ,
- ***CREW***: đọc đồng thời và ghi loại trừ,
- ***ERCW***: đọc loại trừ và ghi đồng thời, và
- ***CRCW***: đọc đồng thời và ghi đồng thời.

(Các chữ viết tắt này thường không được phát âm như là một từ mà là một chuỗi các mẫu tự.)

Trong số các kiểu thuật toán này, các cực—EREW và CRCW—là phổ dụng nhất. Một PRAM chỉ hỗ trợ các thuật toán EREW được gọi là một ***PRAM EREW***, và một PRAM hỗ trợ các thuật toán CRCW được gọi là một ***PRAM CRCW***. Tất nhiên, một PRAM CRCW có thể thi hành các thuật toán EREW, nhưng một PRAM EREW không thể trực tiếp hỗ trợ các truy cập bộ nhớ đồng thời cần có trong các thuật toán CRCW. Phần cứng cơ bản của một PRAM EREW tương đối đơn giản, và do đó chạy nhanh, bởi nó chẳng cần điều quản sự xung đột các lần đọc và ghi bộ nhớ. Một PRAM CRCW yêu cầu nhiều hỗ trợ phần cứng hơn nếu giả thiết thời gian đơn vị phải cung cấp một số đo chính xác hợp lý của khả năng thực hiện thuật toán, nhưng nó cung cấp một mô hình lập trình mà về mặt chứng minh tỏ ra dễ hiểu hơn so với một PRAM EREW.

Trong số hai kiểu thuật toán còn lại—CREW và ERCW—sách giáo khoa thường tập trung nhiều hơn vào CREW. Tuy nhiên, xét theo quan điểm thực tiễn, sự hỗ trợ tính đồng thời cho các lần ghi không khó hơn sự hỗ trợ cho các lần đọc. Trong chương này, ta thường xem một thuật toán như là CRCW nếu nó chứa các lần đọc đồng thời hoặc ghi đồng thời, mà không phân biệt thêm. Ta sẽ đề cập các điểm tinh tế hơn của sự phân biệt này trong Đoạn 30.2.

Khi nhiều bộ xử lý ghi ra cùng vị trí trong một thuật toán CRCW, hiệu ứng của việc ghi song song không được định nghĩa kỹ mà không phải chi tiết hóa thêm. Trong chương này, ta sẽ dùng mẹo ***CRCW chung***: khi vài bộ xử lý ghi vào cùng vị trí bộ nhớ, tất cả chúng phải ghi một giá trị chung. Trong giáo khoa thư, có vài kiểu PRAM khác điều quản bài toán này theo một giả thiết khác. Các chọn lựa khác bao gồm

• ***tùy ý:*** một giá trị tùy ý trong số các giá trị được ghi thực tế được lưu trữ,

• ***mức ưu tiên:*** giá trị được ghi bởi bộ xử lý có chỉ mục thấp nhất được lưu trữ, và

• ***tổ hợp:*** giá trị được lưu trữ là một tổ hợp đãđịnh gồm các giá trị đã ghi.

Trong trường hợp cuối, tổ hợp đãđịnh thường là một hàm giao hoán và kết hợp như phép cộng (lưu trữ tổng của tất cả các giá trị đã ghi) hoặc cực đại (chỉ lưu trữ giá trị cực đại đã ghi).

Đồng bộ hóa và điều khiển

Các thuật toán PRAM phải được đồng bộ hóa cao để làm việc đúng đắn. Tiến trình đồng bộ hóa này được thực hiện ra sao? Ngoài ra, các bộ xử lý trong các thuật toán PRAM thường phải phát hiện sự kết thúc của các điều kiện vòng lặp tùy thuộc vào trạng thái of tất cả các bộ xử lý. Chức năng điều khiển này được thực thi ra sao?

Ta sẽ không đề cập sâu về các vấn đề này. Nhiều máy tính song song thực đã sử dụng một mạng điều khiển nối các bộ xử lý giúp đỡ các điều kiện kết thúc và đồng bộ hóa. Thông thường, mạng điều khiển có thể thực thi các chức năng này nhanh gống như một mạng định tuyến thực thi các tham chiếu bộ nhớ toàn cục.

Với mục tiêu của chúng ta, ta chỉ cần mặc nhận các bộ xử lý vốn dĩ được đồng bộ hóa chặt. Tất cả các bộ xử lý đồng thời thi hành cùng các câu lệnh. Không có bộ xử lý nào chạy lên trước trong khi các bộ xử lý khác tụt lại đàng sau trong mã. Khi đi qua thuật toán song song đầu tiên, ta sẽ chỉ ra nơi mà ta mặc nhận các bộ xử lý được đồng bộ hóa.

Để phát hiện sự kết thúc của một vòng lặp song song tùy thuộc vào trạng thái của tất cả các bộ xử lý, ta sẽ mặc nhận rằng một điều kiện kết thúc song song có thể được kiểm tra qua mạng điều kiện trong $O(1)$ thời gian. Trong giao khoa thư, có vài mô hình PRAM EREW không thực hiện giả thiết này, và thời gian (lôga) để kiểm tra điều kiện vòng lặp phải được gộp trong thời gian thực hiện chung (xem Bài tập 30.1-8). Như sẽ thấy trong Đoạn 30.2,

Các PRAM CRCW không cần một mạng điều khiển để kiểm tra sự kết thúc: chúng có thể phát hiện sự kết thúc của một vòng lặp song song trong $O(1)$ thời gian thông qua việc dùng các lần ghi đồng thời.

Khái quát chương

Đoạn 30.1 giới thiệu kỹ thuật nhảy biến trỏ, cung cấp một cách nhanh để điều tác các danh sách song song. Ta sẽ nêu cách dùng phép nhảy biến trỏ để thực hiện các phép tính tiền tố trên các danh sách và cách thích ứng các thuật toán trên các danh sách để dùng trên các cây nhanh như thế nào. Đoạn 30.2 mô tả năng lực tương đối của các thuật toán CRCW và EREW và chứng tỏ tiến trình truy cập bộ nhớ đồng thời cung cấp năng lực gia tăng.

Đoạn 30.3 trình bày định lý Brent, nêu PRAM có thể mô phỏng hiệu quả các mạch tổ hợp như thế nào. Đoạn này cũng mô tả chủ đề quan trọng về tính hiệu quả công và cung cấp các điều kiện qua đó thuật toán PRAM p-bộ xử lý có thể được phiên dịch hiệu quả thành một thuật toán PRAM p'-bộ xử lý với bất kỳ $p' < p$. Đoạn 30.4 lập lại bài toán thực hiện một phép tính tiền tố trên một danh sách nối kết và nêu cách thức một thuật toán ngẫu nhiên hóa có thể thực hiện phép tính theo cách hiệu quả công. Cuối cùng, Đoạn 30.5 nêu cách dùng một thuật toán tất định để tách tính đối xứng thành song song theo thời gian nhỏ hơn nhiều so với thời gian loga.

Các thuật toán song song trong chương này chủ yếu được rút từ lĩnh vực lý thuyết đồ thị. Chúng chỉ tiêu biểu cho sự lựa chọn vừa đủ về một mảng các thuật toán song song hiện tại. Tuy nhiên, các kỹ thuật giới thiệu trong chương này có thể tiêu biểu cho các kỹ thuật được dùng cho các thuật toán song song trong các lĩnh vực khác của khoa học máy tính.

30.1 Nhảy biến trỏ

Trong số các thuật toán PRAM đáng quan tâm, ta có các thuật toán liên quan đến các biến trỏ. Trong đoạn này, ta nghiên cứu một kỹ thuật mạnh có tên nhảy biến trỏ [pointer jumping], nó cho ra các thuật toán nhanh để vận hành trên các danh sách. Cụ thể, ta giới thiệu a thuật toán $O(\lg n)$ thời gian tính toán khoảng cách đến cuối danh sách cho mỗi đối tượng trong một danh sách n đối tượng. Sau đó, ta sửa đổi thuật toán này

để thực hiện một phép tính "tiền tố song song " trên một danh sách n đối tượng trong $O(\lg n)$ thời gian. Cuối cùng, ta nghiên cứu một kỹ thuật cho phép chuyển đổi nhiều bài toán trên các cây thành các bài toán danh sách, mà như vậy có thể được giải bằng kỹ thuật nhảy biến trỏ. Tất cả các thuật toán trong đoạn này đều là các thuật toán EREW: không cần có các truy cập đồng thời đến bộ nhớ toàn cục.

30.1.1 Xếp hạng danh sách

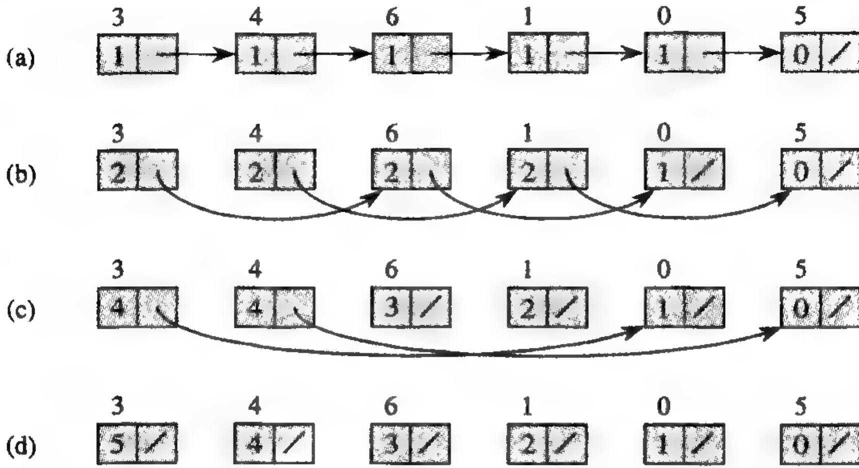
Thuật toán song song đầu tiên của chúng ta hoạt động trên các danh sách. Ta có thể lưu trữ một danh sách trong một PRAM giống như lưu trữ các danh sách trong một RAM bình thường. Tuy nhiên, để hoạt động song song trên các đối tượng danh sách, ta nên gán cho mỗi đối tượng một bộ xử lý "chịu trách nhiệm." Ta mặc nhận có các bộ xử lý với số lượng ngang bằng các đối tượng, và bộ xử lý thứ i chịu trách nhiệm đối tượng thứ i. Ví dụ, Hình 30.2(a) nêu một danh sách nối kết bao gồm dãy đối tượng <3, 4, 6, 1, 0, 5>. Bởi có một bộ xử lý cho mỗi đối tượng danh sách, nên mọi đối tượng trong danh sách có thể được vận hành trên theo bộ xử lý chịu trách nhiệm củanó trong $O(1)$ thời gian.

Giả sử ta có một danh sách nối kết đơn L với n đối tượng và muốn tính toán, với mỗi đối tượng trong L, khoảng cách củanó từ cuối danh sách. Chính thức hơn, nếu *next* là trường biến trỏ, ta muốn tính toán một giá trị $d[i]$ với mỗi đối tượng i trong danh sách sao cho

$$d[i] = \begin{cases} 0 & \text{nếu } next[i] = \text{NIL}, \\ d[next[i]] + 1 & \text{nếu } next[i] \neq \text{NIL}. \end{cases}$$

Ta gọi bài toán tính toán các giá trị d là ***bài toán xếp hạng danh sách*** [list-ranking].

Một giải pháp cho bài toán xếp hạng danh sách đó là chỉ việc lan truyền các khoảng cách trở lại từ cuối danh sách. Phương pháp này mất $\Theta(n)$ thời gian, bởi đối tượng thứ k từ cuối phải đợi các đối tượng $k - 1$ theo nó để xác định các khoảng cách của chúng từ cuối trước khi nó có thể xác định khoảng cách riêng của nó. Giải pháp này chủ yếu là một thuật toán nối tiếp.



Hình 30.2 Tìm khoảng cách từ mỗi đối tượng trong một danh sách *n* đối tượng đến cuối danh sách trong Θ(lg *n*) thời gian dùng kỹ thuật nhảy biến trỏ. **(a)** Một danh sách nối kết được biểu diễn trong một PRAM bằng các giá trị *d* được khởi tạo. Vào cuối thuật toán, mỗi giá trị *d* lưu giữ khoảng cách của đối tượng của nó từ cuối danh sách. Bộ xử lý chịu trách nhiệm của mỗi đối tượng xuất hiện bên trên đối tượng. **(b)-(d)** Các biến trỏ và các giá trị *d* sau mỗi lần lặp lại của vòng lặp **while** trong thuật toán LIST RANK.

Một giải pháp song song hiệu quả, chỉ yêu cầu $O(\lg n)$ thời gian, được căn cứ vào mã giả song song dưới đây.

LIST-RANK(*L*)

1 **for** mỗi bộ xử lý *i*, song song
2   **do if** *next*[*i*] = NIL
3   **then** *d*[*i*] ← 0
4   **else** *d*[*i*] ← 1
5 **while** ở đó tồn tại một đối tượng *i* sao cho sao cho *next*[*i*] ≠ NIL
6   **do for** mỗi bộ xử lý *i*, song song
7   **do if** *next*[*i*] ≠ NIL
8   **then** *d*[*i*] ← *d*[*i*] + *d*[*next*[*i*]]
9   *next*[*i*] ← *next*[*next*[*i*]]

Hình 30.2 nêu cách thức mà thuật toán tính toán các khoảng cách. Mỗi phần của hình nêu trạng thái của danh sách trước một lần lặp lại của vòng lặp **while** của các dòng 5-9. Phần (a) nêu danh sách ngay sau khi khởi tạo. Trong lần lặp lại đầu tiên, 5 đối tượng danh sách đầu tiên có các biến trỏ phi NIL, sao cho các dòng 8-9 được thi hành bởi các bộ xử lý chịu trách nhiệm của chúng. Kết quả xuất hiện trong phần (b) của

hình. Trong lần lặp lại thứ hai, chỉ 4 đối tượng đầu tiên có các biến trỏ phi NIL; kết quả của lần lặp lại này được nêu trong phần (c). Trong lần lặp lại thứ ba, chỉ 2 đối tượng đầu tiên được vận hành, và phần (d) nêu kết quả chung cuộc, ở đó tất cả các đối tượng đều có các biến trỏ NIL.

Ý tưởng mà dòng 9 thực thi, ở đó ta ấn định $next[i] \leftarrow next[next[i]]$ với tất cả các biến trỏ phi NIL $next[i]$, được gọi là ***nhảy biến trỏ***. Lưu ý, kỹ thuật nhảy biến trỏ đã thay đổi các trường biến trỏ, như vậy hủy cấu trúc của danh sách. Nếu phải bảo toàn cấu trúc danh sách, ta tạo các bản sao của các biến trỏ *next* và dùng bản sao để tính toán các khoảng cách.

Tính đúng đắn

LIST-RANK duy trì sự bất biến rằng tại đầu mỗi lần lặp lại của vòng lặp **while** trong các dòng 5-9, với mỗi đối tượng i, nếu ta cộng các giá trị d trong danh sách con có i dẫn đầu, ta sẽ được khoảng cách đúng đắn từ i đến cuối danh sách ban đầu L. Ví dụ, trong Hình 30.2(b), danh sách con có đối tượng 3 dẫn đầu là dãy <3, 6, 0> mà các giá trị d của nó 2, 2, và 1 tổng cộng thành 5, khoảng cách củanó từ cuối danh sách ban đầu. Sở dĩ sự bất biến được duy trì đó là vì khi mỗi đối tượng "nối khử" phần tử kế vị của nó trong danh sách, nó cộng giá trị d của phần tử kế vị của nó vào chính nó.

Nhận thấy để thuật toán nhảy biến trỏ này làm việc đúng đắn, các lần truy cập bộ nhớ song song phải được đồng bộ hóa. Mỗi lần thi hành của dòng 9 có thể cập nhật vài biến trỏ *next*. Ta dựa vào tất cả các lần đọc bộ nhớ bên phía phải của phép gán (reading $next[next[i]]$) xảy ra trước mọi lần ghi của bộ nhớ (writing $next[i]$) bên phía trái.

Giờ đây, ta hãy xem tại sao LIST-RANK là một thuật toán EREW. Bởi mỗi bộ xử lý chịu trách nhiệm tối đa một đối tượng, nên mọi lần đọc và ghi trong các dòng 2-7 là loại trừ, giống như các lần ghi trong các dòng 8-9. Nhận thấy kỹ thuật nhảy biến trỏ duy trì sự bất biến rằng với hai đối tượng riêng biệt bất kỳ i và j, hoặc $next[i] \neq next[j]$ hoặc $next[i] = next[j] =$ NIL. Sự bất biến này chắn chắn là đúng với danh sách ban đầu, và nó được duy trì bằng dòng 9. Bởi tất cả các giá trị *next* phi NIL đều riêng biệt, tất cả các lần đọc trong dòng 9 đều loại trừ.

Ta phải mặc nhận thực hiện một dạng đồng bộ hóa nào đó trong dòng 8 nếu tất cả các lần đọc sẽ là loại trừ. Nói cụ thể, ta yêu cầu tất cả các bộ xử lý i đọc $d[i]$ rồi $d[next[i]]$. Với dạng đồng bộ hóa này, nếu một đối tượng i có $next[i] \neq$ NIL và có một đối tượng j khác trỏ đến i (nghĩa là, $next[j] = i$), thì lần đọc đầu tiên truy nạp $d[i]$ cho bộ xử lý i và lần đọc thứ hai truy nạp $d[i]$ cho bộ xử lý j. Như vậy, LIST-RANK là một thuật toán EREW.

Từ đây trở đi, ta bỏ qua các chi tiết đồng bộ hóa như vậy và mặc nhận rằng PRAM và môi trường lập trình mã giả của nó tác động theo cách đồng bộ hóa, nhất quán, với tất cả các bộ xử lý thi hành các lần đọc và ghi cùng một lúc.

Phân tích

Giờ đây ta chứng tỏ nếu có n đối tượng trong danh sách L, thì LIST-RANK mất $O(\lg n)$ thời gian. Bởi tiến trình khởi tạo mất $O(1)$ thời gian và mỗi lần lặp lại của vòng lặp **while** mất $O(1)$ thời gian, nên ta chỉ cần chứng tỏ có chính xác $\lceil \lg n \rceil$ lần lặp lại. Nhận xét chính đó là mỗi bước của kỹ thuật nhảy biến trỏ sẽ biến đổi từng danh sách thành hai danh sách xen kẽ: một danh sách bao gồm các đối tượng trong các vị trí chẵn và danh sách kia bao gồm các đối tượng trong các vị trí lẻ. Như vậy, mỗi bước nhảy biến trỏ sẽ nhân đôi số lượng danh sách và chia đôi các chiều dài của chúng. Do đó, vào cuối $\lceil \lg n \rceil$ lần lặp lại, tất cả các danh sách chỉ chứa một đối tượng.

Ta đang mặc nhận đợt kiểm tra sự kết thúc trong dòng 5 mất $O(1)$ thời gian, có lẽ là do một mạng điều khiển trong PRAM EREW. Bài tập 30.1-8 yêu cầu bạn mô tả một thực thi EREW $O(\lg n)$ thời gian của LIST-RANK thực hiện đợt kiểm tra sự kết thúc một cách rõ rệt trong mã giả.

Ngoài thời gian thực hiện song song, ta có một số đo khả năng thực hiện đáng quan tâm khác cho các thuật toán song song. Ta định nghĩa ***công*** [work] mà một thuật toán song song thực hiện dưới dạng tích thời gian thực hiện của nó và số lượng bộ xử lý mà nó yêu cầu. Theo trực giác, công là lượng tính toán mà một RAM nối tiếp thực hiện khi nó mô phỏng thuật toán song song.

Thủ tục LIST-RANK thực hiện $\Theta(n \lg n)$ công, bởi nó yêu cầu n bộ xử lý và chạy trong $\Theta(\lg n)$ thời gian. Thuật toán nối tiếp đơn giản cho bài toán xếp hạng danh sách chạy trong $\Theta(n)$ thời gian, nêu rõ LIST-RANK thực hiện nhiều công hơn so với mức tuyệt đối cần thiết, nhưng chỉ theo một thừa số lôga.

Ta định nghĩa một thuật toán PRAM A là ***hiệu quả công*** [work-efficient] đối với một thuật toán (nối tiếp hoặc song song) B cho cùng bài toán nếu công mà A thực hiện nằm trong một thừa số bất biến của công mà B thực hiện. Ta cũng nói một cách đơn giản hơn rằng một thuật toán PRAM A là ***hiệu quả công*** nếu nó hiệu quả công đối với thuật toán khả dĩ tốt nhất trên một RAM nối tiếp. Bởi thuật toán nối tiếp khả dĩ tốt nhất để xếp hạng danh sách chạy trong $\Theta(n)$ thời gian trên một RAM nối tiếp, nên LIST-RANK không phải là hiệu quả công. Ta sẽ trình bày một thuật toán song song hiệu quả công để xếp hạng danh sách trong Đoạn 30.4.

30.1.2 Tiền tố song song trên một danh sách

Kỹ thuật nhảy biến trỏ mở rộng vượt quá ứng dụng xếp hạng danh sách. Trong ngữ cảnh các mạch số học, Đoạn 29.2.2 nêu cách dùng phép tính "tiền tố" để thực hiện phép cộng nhị phân nhanh chóng. Giờ đây, ta nghiên cứu cách dùng kỹ thuật nhảy biến trỏ để thực hiện các phép tính tiền tố. Thuật toán EREW của chúng ta cho bài toán tiền tố chạy trong $O(\lg n)$ thời gian trên các danh sách n-đối tượng.

Một ***phép tính tiền tố*** [prefix computation] được định nghĩa theo dạng một toán tử kết hợp, nhị phân $\otimes$. Phép tính nhận một dãy $<x_1, x_2, ..., x_n>$ làm đầu vào và kết xuất một dãy $<y_1, y_2, ..., y_n>$ sao cho $y_1 = x_1$ và

$$\begin{aligned} y_k &= y_{k-1} \otimes x_k \\ &= x_1 \otimes x_2 \otimes \ldots \otimes x_k \end{aligned}$$

với $k = 2, 3,..., n$. Nói cách khác, mỗi y_k có được bằng cách "nhân" k thành phần đầu tiên của x_k dãy xuất khẩu với nhau—do đó, được gọi là "tiền tố." (Định nghĩa trong Chương 29 lập chỉ số các dãy từ 0, trong khi đó định nghĩa này lập chỉ số từ 1—một sự khác biệt không cần thiết.)

Để lấy ví dụ của một phép tính tiền tố, giả sử mọi thành phần của một danh sách n-đối tượng chứa giá trị 1, và cho $\otimes$ là phép cộng bình thường. Bởi thành phần thứ k của danh sách chứa giá trị $x_k = 1$ với $k = 1, 2,..., n$, một phép tính tiền tố tạo ra $y_k = k$, chỉ số của thành phần thứ k. Như vậy, một cách khác để tiến hành xếp hạng danh sách đó là đảo danh sách (có thể được thực hiện trong $O(1)$ thời gian), thực hiện phép tính tiền tố này, và trừ 1 ra khỏi mỗi giá trị đã tính toán.

Giờ đây ta nêu cách thức mà một thuật toán EREW có thể tính toán các tiền tố song song trong $O(\lg n)$ thời gian trên các danh sách n-đối tượng. Để tiện dụng, ta định nghĩa hệ ký hiệu

$$[i, j] = x_i \otimes x_{i+1} \otimes \ldots \otimes x_j$$

với các số nguyên i và j trong miền $1 \le i \le j \le n$. Vậy, $[k, k] = x_k$ với $k = 1, 2,..., n$, và

$$[i, k] = [i, j] \otimes [j + 1, k]$$

với $0 \le i \le j < k \le n$. Theo dạng hệ ký hiệu này, mục tiêu của một phép tính tiền tố đó là tính toán $y_k = [1, k]$ với $k = 1, 2,..., n$.

Khi thực hiện một phép tính tiền tố trên một danh sách, ta muốn xác định thứ tự của dãy đầu vào $<x_1, x_2,..., x_n>$ theo cách nối kết các đối tượng với nhau trong danh sách, và chứ không phải theo chỉ số của đối tượng trong mảng của bộ nhớ lưu trữ các đối tượng. (Bài tập 30.1-2 yêu cầu một thuật toán tiền tố cho các mảng.) Thuật toán EREW dưới đây

bắt đầu bằng một giá trị $x[i]$ trong mỗi đối tượng i của một danh sách L. Nếu đối tượng i là đối tượng thứ k từ đầu danh sách, thì $x[i] = x_k$ là thành phần thứ k của dãy đầu vào. Như vậy, phép tính tiền tố song song tạo ra $y[i] = y_k = [1, k]$.

LIST-PREFIX(L)

1 **for** mỗi bộ xử lý i, song song

2   **do** $y[i] \leftarrow x[i]$

3 **while** ở đó tồn tại một đối tượng i sao cho $next[i] \neq$ NIL

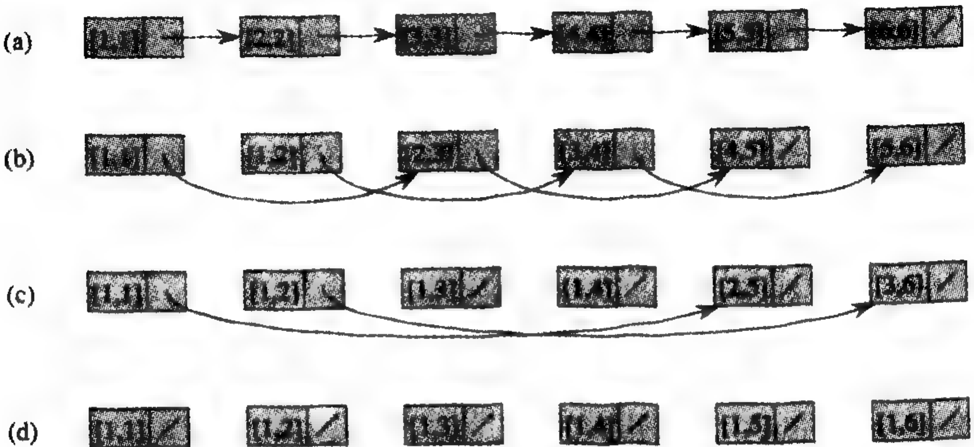
4   **do for** mỗi bộ xử lý i, song song

5   **do if** $next[i] \neq$ NIL

6   **then** $y[next[i]] \leftarrow y[i] \otimes y[next[i]]$

7 $next[i] \leftarrow next[next[i]]$

Mã giả và Hình 30.3 cho thấy tính tương tự giữa thuật toán này và LIST-RANK. Những điểm khác biệt duy nhất đó là tiến trình khởi tạo và cập nhật của các giá trị d hoặc y. Trong LIST-RANK, bộ xử lý i cập nhật $d[i]$—giá trị d riêng củanó—trong khi đó trong LIST-PREFIX, bộ xử lý i cập nhật $y[next[i]]$—giá trị y của một bộ xử lý khác. Lưu ý, LIST-PREFIX là EREW với cùng lý do như LIST-RANK: kỹ thuật nhảy biến trỏ duy trì sự bất biến rằng với các đối tượng riêng biệt i và j, hoặc $next[i] \neq next[j]$ hoặc $next[i] = next[j] =$ NIL.



Hình 30.3 Thuật toán tiền tố song song LIST-PREFIX trên một danh sách nối kết. **(a)** Giá trị y ban đầu của đối tượng thứ k trong danh sách là $[k, k]$. Biến trỏ *next* của đối tượng thứ k trỏ đến đối tượng thứ $(k + 1)$, hoặc NIL với đối tượng cuối. **(b)-(d)** Các giá trị y và *next* trước mỗi đợt kiểm tra trong dòng 3. Đáp án chung cuộc được nêu trong phần (d), ở đó giá trị y của đối tượng thứ k là $[1, k]$ với tất cả k.

Hình 30.3 nêu trạng thái của danh sách trước mỗi lần lặp lại của vòng lặp **while**. Thủ tục duy trì sự bất biến rằng tại cuối lần thi hành thứ t của vòng lặp **while**, bộ xử lý thứ k lưu trữ $[\max(1, k - 2^t + 1), k]$, với $k = 1, 2,..., n$. Trong lần lặp lại đầu tiên, đối tượng danh sách thứ k thoạt đầu trỏ đến đối tượng thứ $(k + 1)$, ngoại trừ đối tượng chót có một biến trỏ NIL. Dòng 6 khiến đối tượng thứ k, với $k = 1, 2,..., n - 1$, truy nạp giá trị $[k + 1, k + 1]$ từ phần tử kế vị củanó. Sau đó, nó thực hiện phép toán $[k, k] \otimes [k + 1, k + 1]$, cho ra $[k, k + 1]$, mà nó lưu trữ trở lại vào phần tử kế vị của nó. Như vậy, các biến trỏ *next* được nhảy như trong LIST-RANK, và kết quả của lần lặp lại đầu tiên xuất hiện trong Hình 30.3(b). Ta có thể xem lần lặp lại thứ hai tương tự như vậy. Với $k = 1, 2,..., n - 2$, đối tượng thứ k truy nạp giá trị $[k + 1, k + 2]$ từ phần tử kế vị của nó (như đã định nghĩa bởi giá trị mới trong trường *next* của nó), rồi nó lưu trữ $[k - 1, k] \otimes [k + 1, k + 2] = [k - 1, k + 2]$ vào phần tử kế vị của nó. Kết quả được nêu trong Hình 30.3(c). Trong lần lặp lại thứ ba và cuối cùng, chỉ hai đối tượng danh sách đầu tiên có các biến trỏ phi NIL, và chúng truy nạp các giá trị từ các phần tử kế vị của chúng trong các danh sách tương ứng của chúng. Kết quả chung cuộc xuất hiện trong Hình 30.3(d). Nhận xét chính để LIST-PREFIX làm việc đó là tại mỗi bước, nếu ta thực hiện một phép tính tiền tố trên mỗi trong số vài danh sách hiện có, mỗi đối tượng sẽ đạt được giá trị đúng đắn của nó.

Bởi hai thuật toán dùng cùng cơ chế nhảy biến trỏ, nên LIST-PREFIX có cùng kiểu phân tích như LIST-RANK: thời gian thực hiện là $O(\lg n)$ trên một PRAM EREW, và tổng công được thực hiện là $\Theta(n \lg n)$.

30.1.3 Kỹ thuật tua Euler

Trong đoạn này, ta sẽ giới thiệu kỹ thuật tua Euler và nêu cách áp dụng nó cho bài toán tính toán độ sâu của mỗi nút trong một cây nhị phân n-nút. Phép tính tiền tố song song chính là một bước chính trong thuật toán EREW $O(\lg n)$ thời gian này.

Để lưu trữ các cây nhị phân trong một PRAM, ta dùng một phép biểu diễn cây nhị phân đơn giản của kiểu sắp xếp đã trình bày trong Đoạn 11.4. Mỗi nút i có các trường *parent*[i], *left*[i], và *right*[i], trỏ đến cha, con trái, và con phải của nút i, theo thứ tự nêu trên. Ta hãy mặc nhận rằng mỗi nút được định danh bởi một số nguyên không âm. Với các lý do sẽ nêu dưới đây, ta phối hợp không phải là một mà là ba bộ xử lý với mỗi nút; ta gọi chúng là các bộ xử lý A, B, và C của nút. Ta phải có khả năng ánh xạ dễ dàng giữa một nút và ba bộ xử lý của nó; ví dụ, nút i có thể kết hợp với các bộ xử lý $3i$, $3i + 1$, và $3i + 2$.

Tính toán độ sâu của mỗi nút trong một cây n-nút mất $O(n)$ thời gian trên một RAM nối tiếp. Một thuật toán song song đơn giản để tính toán

các độ sâu sẽ lan truyền một "sóng" đổ xuống từ gốc của cây. Sóng đụng tất cả các nút có cùng độ sâu cùng một lúc, và như vậy bằng cách gia số một bộ đếm được mang đi cùng với sóng, ta có thể tính toán độ sâu của mỗi nút. Thuật toán song song này làm việc tốt trên một cây nhị phân hoàn chỉnh, bởi nó chạy trong thời gian tỷ lệ với chiều cao của cây. Tuy nhiên, chiều cao của cây có thể lớn tới mức $n - 1$, trong trường hợp đó thuật toán sẽ chạy trong $\Theta(n)$ thời gian—không tốt hơn thuật toán nối tiếp. Tuy nhiên, dùng kỹ thuật tua Euler [Euler tour], ta có thể tính toán các độ sâu nút trong $O(\lg n)$ thời gian trên một PRAM EREW, bất luận chiều cao của cây.

Một ***tua Euler*** của một đồ thị là một chu trình băng ngang mỗi cạnh chính xác một lần, mặc dù nó có thể ghé một đỉnh nhiều lần. Qua Bài toán 23-3, một đồ thị có hướng, liên thông, có một tua Euler nếu và chỉ nếu với tất cả các đỉnh v, độ-vào của v bằng độ-ra của v. Bởi mỗi cạnh không hướng (u, v) trong một đồ thị không hướng sẽ ánh xạ theo hai cạnh có hướng (u, v) và (v, u) trong phiên bản có hướng, phiên bản có hướng của bất kỳ đồ thị nào không hướng, liên thông—và do đó của bất kỳ cây không hướng nào—có một tua Euler.

Để tính toán các độ sâu của các nút trong một cây nhị phân T, trước tiên ta hình thành một tua Euler của phiên bản có hướng của T (được xem là một đồ thị không hướng). Tua tương ứng với một tầng của cây và được biểu thị trong Hình 30.4(a) bằng một danh sách nối kết chạy qua các nút của cây. Cấu trúc của nó như sau:

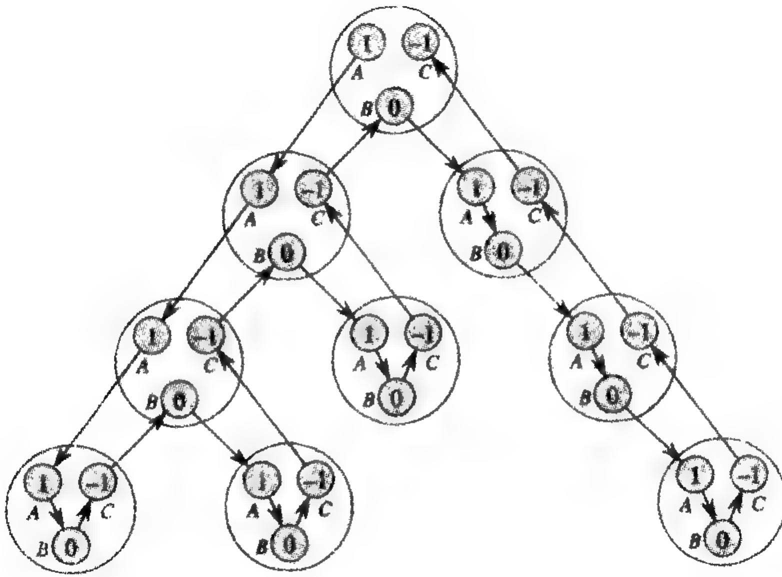
- • Bộ xử lý A của một nút trỏ đến bộ xử lý A của con trái của nó, nếu nó tồn tại, và bằng không đến bộ xử lý B của riêng nó.

- • Bộ xử lý B của một nút trỏ đến bộ xử lý A của con phải của nó, nếu nó tồn tại, và bằng không đến bộ xử lý C của riêng nó.

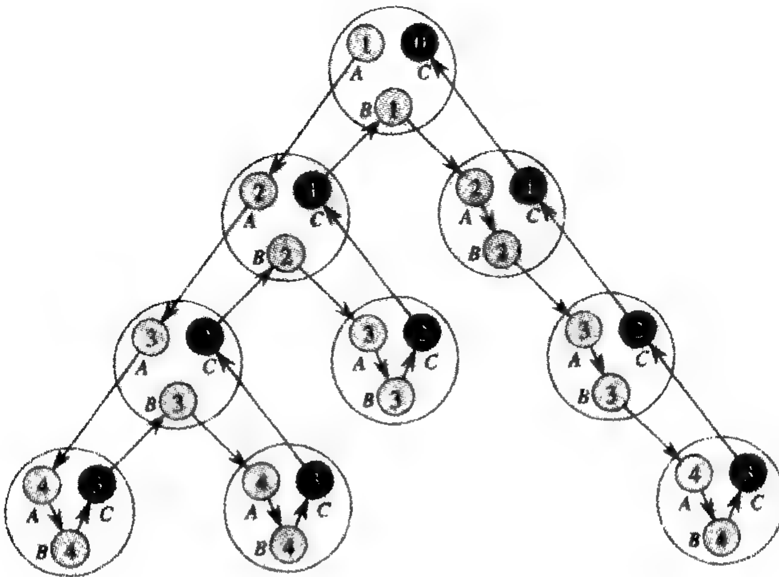
- • Bộ xử lý C của một nút trỏ đến bộ xử lý B của cha nó nếu nó là một con trái và đến bộ xử lý C của cha nó nếu nó là một con phải. bộ xử lý C của gốc trỏ đến NIL.

Như vậy, đầu danh sách nối kết được lập thành bởi tua Euler là bộ xử lý A của gốc, và đuôi là bộ xử lý C của gốc. Cho các biến trỏ bao gồm cây ban đầu, một tua Euler có thể được kiến tạo trong $O(1)$ thời gian.

Sau khi có danh sách nối kết biểu thị tua Euler của T, ta đặt một 1 trong mỗi bộ xử lý A, một 0 trong mỗi bộ xử lý B, và một -1 trong mỗi bộ xử lý C, như đã nêu trong Hình 30.4(a). Sau đó, ta thực hiện một song song phép tính tiền tố dùng phép cộng bình thường làm phép toán kết hợp, như đã làm trong Đoạn 30.1.2. Hình 30.4(b) nêu kết quả của phép tính tiền tố song song.



(a)



(b)

Hình 30.4 Dùng kỹ thuật tua Euler để tính toán độ sâu của mỗi mắt trong một cây nhị phân. **(a)** Tua Euler là một danh sách tương ứng với một tầng của cây. Mỗi bộ xử lý chứa một số được một phép tính tiền tố song song sử dụng để tính toán các độ sâu mắt. **(b)** Kết quả của phép tính tiền tố song song trên danh sách nối kết từ (a). Bộ xử lý *C* của mỗi mắt (được tô đen) chứa độ sâu của mắt. (Bạn có thể xác minh kết quả của phép tính tiền tố này bằng cách tính toán nó theo nối tiếp.)

Ta biện luận rằng sau khi thực hiện phép tính tiền tố song song, độ sâu của mỗi nút thường trú trong bộ xử lý C của nút. Tại sao? Các con số được đặt vào các bộ xử lý A, B, và C theo cách sao cho hiệu ứng thuần của việc ghé thăm một cây con sẽ là cộng 0 vào tổng tới [running total]. Bộ xử lý A của mỗi nút i đóng góp 1 vào tổng tới trong cây con trái của i, phản ánh độ sâu con trái của i là một độ sâu lớn hơn độ sâu của i. Bộ xử lý B đóng góp 0 bởi độ sâu của con trái của nút i bằng độ sâu của con phải của nút i. Bộ xử lý C đóng góp -1, sao cho từ góc nhìn của cha của nút i, nguyên cả lượt ghé thăm cây con có gốc tại nút i không có hiệu ứng gì trên tổng tới.

Danh sách biểu thị tua Euler có thể được tính toán trong $O(1)$ thời gian. Nó có $3n$ đối tượng, và như vậy phép tính tiền tố song song chỉ mất $O(\lg n)$ thời gian. Như vậy, tổng thời lượng để tính toán tất cả các độ sâu nút là $O(\lg n)$. Bởi không cần các lần truy cập bộ nhớ đồng thời, nên thuật toán là một thuật toán EREW.

Bài tập

30.1-1

Nêu một thuật toán EREW $O(\lg n)$-thời gian xác định xem với mỗi đối tượng trong một danh sách n-đối tượng nó có phải là đối tượng giữa (thứ $\lfloor n/2 \rfloor$).

30.1-2

Nêu một thuật toán EREW $O(\lg n)$-thời gian để thực hiện phép tính tiền tố trên một mảng $x[1..n]$. Đừng dùng các biến trỏ, nhưng thực hiện trực tiếp các phép tính chỉ số.

30.1-3

Giả sử mỗi đối tượng trong một danh sách n-đối tượng L được tô màu đỏ hoặc xanh dương. Nêu một thuật toán EREW hiệu quả để tạo thành hai danh sách từ các đối tượng trong L: một bao gồm các đối tượng xanh dương và một bao gồm các đối tượng đỏ.

30.1-4

Một PRAM EREW có n đối tượng được phân phối giữa vài danh sách vòng rời nhau. Nêu một thuật toán hiệu quả xác định một đối tượng đại diện tùy ý cho mỗi danh sách và cho mỗi đối tượng trong danh sách biết đồng nhất thức của đại diện. Mặc nhận mỗi bộ xử lý biết chỉ số duy nhất riêng của nó.

30.1-5

Nêu một thuật toán EREW $O(\lg n)$-thời gian để tính toán kích cỡ của cây con có gốc tại mỗi nút của một cây nhị phân n-nút. (*Mách nước:* Lấy hiệu của hai giá trị trong một tổng tới [running sum] dọc theo một tua Euler.)

30.1-6

Nêu một thuật toán EREW hiệu quả để tính toán các kiểu đánh số tiền cấp [preorder], tại cấp [inorder], và hậu cấp [postorder] cho một cây nhị phân tùy ý.

30.1-7

Mở rộng kỹ thuật tua Euler từ các cây nhị phân đến các cây có thứ tự với các độ nút tùy ý. Cụ thể, hãy mô tả một phần biểu diễn cho các cây có thứ tự cho phép áp dụng kỹ thuật tua Euler. Nêu một thuật toán EREW để tính toán các độ sâu nút của một cây có thứ tự n-nút trong $O(\lg n)$ thời gian.

30.1-8

Mô tả một thực thi $O(\lg n)$-thời gian của LIST-RANK thực hiện rõ rệt đợt kiểm tra kết thúc vòng lặp. (*Mách nước:* Xen kẽ đợt kiểm tra với thân vòng lặp.)

30.2 Các thuật toán CRCW và các thuật toán EREW

Cuộc tranh luận về việc phần cứng của một máy tính song song có phải cung cấp các đợt truy cập bộ nhớ đồng thời hay không quả là một cuộc tranh luận hỗn độn. Một số lập luận rằng các cơ chế phần cứng để hỗ trợ các thuật toán CRCW quả quá tốn kém và hiếm khi được dùng nên khó lòng biện hộ. Những người khác lại phàn nàn các PRAM EREW cung cấp một mô hình lập trình quá hạn chế. Đáp án cho cuộc tranh luận này rơi vào khoảng lưng chừng, và người ta đã đề xuất nhiều mô hình thỏa hiệp khác nhau. Tuy vậy, ta cũng nên xét các lần truy cập bộ nhớ đồng thời cung cấp những ưu điểm thuật toán nào.

Trong đoạn này, ta sẽ chứng tỏ có các bài toán mà một thuật toán CRCW làm tốt hơn so với thuật toán EREW khả dĩ tốt nhất. Với bài toán tìm các đồng nhất thức của các gốc của các cây trong một rừng, các lần đọc đồng thời cho phép một thuật toán nhanh hơn. Với bài toán tìm thành phần cực đại trong một mảng, các lần ghi đồng thời cho phép một nhanh hơn thuật toán.

Một bài toán ở đó các lần đọc đồng thời có thể trợ giúp

Giả sử ta có một rừng các cây nhị phân ở đó mỗi nút *i* có một biến trỏ *parent*[*i*] đến cha của nó, và ta muốn mỗi nút để tìm ra đồng nhất thức của gốc cây của nó. Kết hợp bộ xử lý *i* với mỗi nút *i* trong một rừng *F*, thuật toán nhảy biến trỏ dưới đây lưu trữ đồng nhất thức của gốc cây của mỗi nút *i* trong *root*[*i*].

```
FIND-ROOTS(F)
1 for mỗi bộ xử lý i, song song
2       do if parent[i] = NIL
3             then root[i] ← i
4 while ở đó tồn tại một nút i sao cho parent[i] ≠ NIL
5       do for mỗi bộ xử lý i, song song
6             do if parent[i] ≠ NIL
7                   then root[i] ← root[parent[i]]
8 parent[i] ← parent[parent[i]]
```

Hình 30.5 minh họa phép toán của thuật toán này. Sau khi các dòng 1-3 thực hiện tiến trình khởi tạo, xem Hình 30.5(a), chỉ những nút nào biết các đồng nhất thức của các gốc của chúng mới chính là các gốc. Vòng lặp **while** của các dòng 4-8 thực hiện bước nhảy biến trỏ và điền vào các trường *root*.

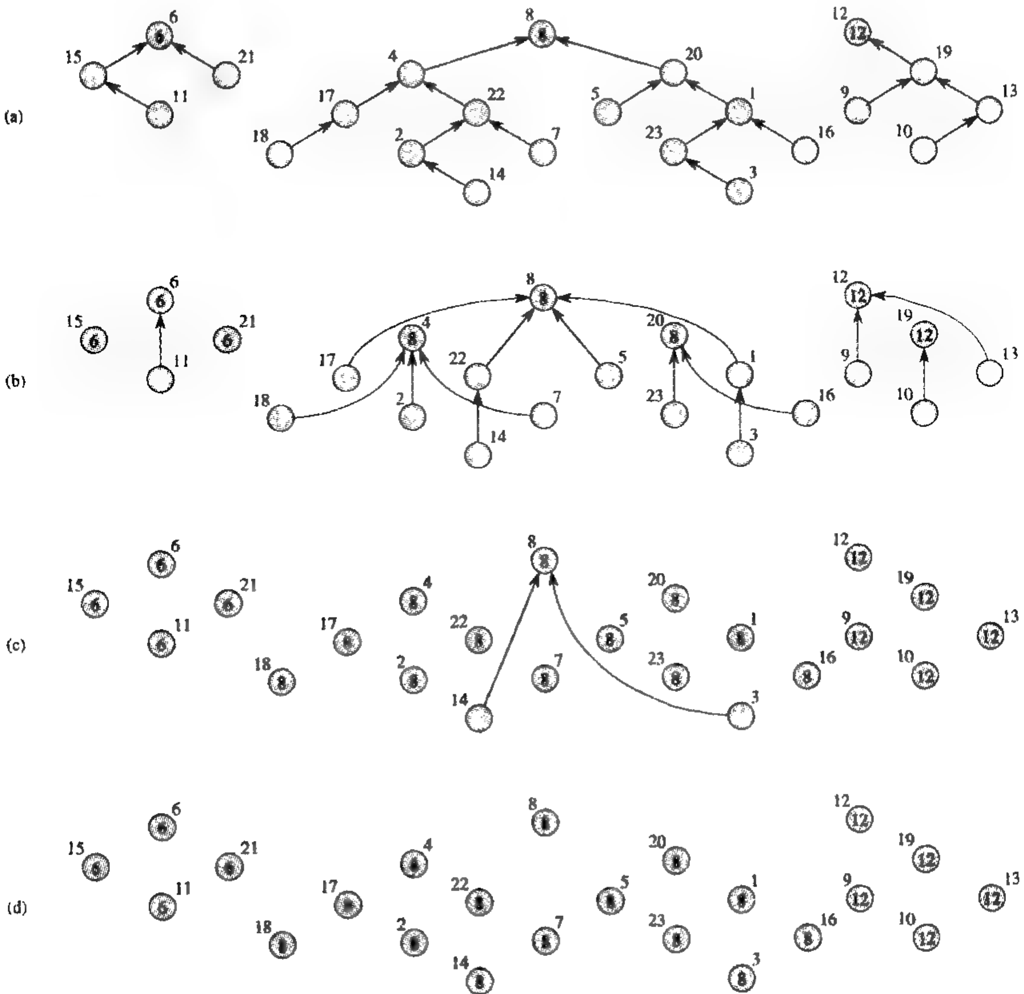
Các Hình 30.5(b)-(d) nêu trạng thái của rừng sau lần lặp lại đầu tiên, thứ hai, và thứ ba của vòng lặp. Như có thể thấy, thuật toán duy trì sự bất biến rằng nếu *parent*[*i*] = NIL, thì *root*[*i*] đã được gán đồng nhất thức của gốc của nút.

Ta biện luận rằng FIND-ROOTS là một thuật toán CREW chạy trong $O(\lg d)$ thời gian, ở đó d là độ sâu của cây độ sâu cực đại trong rừng. Các lần ghi duy nhất xảy ra trên các dòng 3, 7, và 8, và tất cả chúng là loại trừ bởi trong mỗi lần ghi, bộ xử lý *i* chỉ ghi vào nút *i*. Tuy nhiên, các lần đọc trong các dòng 7-8 là đồng thời, bởi một số nút có thể có các biến trỏ đến cùng nút. Ví dụ, trong Hình 30.5(b), ta thấy điều đó trong lần lặp lại thứ hai của vòng lặp **while**, *root*[4] và *parent*[4] được đọc bởi các bộ xử lý 18, 2, và 7.

Thời gian thực hiện của FIND-ROOTS là $O(\lg d)$ với chủ yếu cùng lý do như trường hợp LIST-RANK: chiều dài của mỗi lộ trình được chia đôi trong mỗi lần lặp lại. Hình 30.5 nêu đặc tính này một cách rõ ràng.

n nút trong một rừng có thể xác định các gốc của các cây nhị phân của chúng chỉ dùng các lần đọc loại trừ nhanh như thế nào? Một lập

luận đơn giản chứng tỏ cần $\Omega(\lg n)$ thời gian. Nhận xét chính đó là khi các lần đọc là loại trừ, mỗi bước của PRAM cho phép một mẩu thông tin đã cho được chép ra tối đa một vị trí bộ nhớ khác; như vậy số lượng vị trí có thể chứa một mẩu thông tin đã cho sẽ nhân đôi tối đa theo mỗi bước. Xem xét một cây đơn, thoạt đầu ta có tối đa 1 vị trí bộ nhớ lưu trữ đồng nhất thức của gốc. Sau bước 1, có tối đa 2 vị trí có thể chứa đồng nhất thức của gốc; sau k bước, có tối đa 2^{k-1} vị trí có thể chứa đồng nhất thức của gốc. Nếu kích cỡ của cây là $\Theta(n)$, ta cần $\Theta(n)$ vị trí để chứa đồng nhất thức của gốc khi thuật toán kết thúc; như vậy, yêu cầu tất cả $\Omega(\lg n)$ bước.



Hình 30.5 Tìm các gốc trong một rừng các cây nhị phân trên một PRAM CREW. Các số nút nằm cạnh các nút, và các trường *root* đã kết xuất xuất hiện trong các nút. Các nối kết biểu thị cho các biến trỏ *parent*. **(a)-(d)** Trạng thái của các cây trong rừng mỗi lần dòng 4 của FIND-ROOTS được thi hành. Lưu ý, các chiều dài lộ trình được chia đôi trong mỗi lần lặp lại.

Mỗi khi độ sâu d của cây độ sâu cực đại trong rừng là $2^{o(\lg n)}$, thuật toán CREW FIND-ROOTS theo tiệm cận làm tốt hơn bất kỳ thuật toán EREW nào. Cụ thể, với một rừng n-nút bất kỳ có cây độ sâu cực đại là một cây nhị phân cân đối với $\Theta(n)$ nút, $d = O(\lg n)$, trong trường hợp đó FIND-ROOTS chạy trong $O(\lg \lg n)$ thời gian. Mọi thuật toán EREW cho bài toán này phải chạy trong $\Omega(\lg n)$ thời gian, mà theo tiệm cận là chậm hơn. Như vậy, các lần đọc đồng thời sẽ giúp cho bài toán này. Bài tập 30.2-1 đưa ra một bối cảnh đơn giản hơn ở đó các lần đọc đồng thời có thể hỗ trợ.

Một bài toán ở đó các lần ghi đồng thời hỗ trợ

Để chứng minh các lần ghi đồng thời có ưu điểm về khả năng thực hiện hơn so với các lần ghi loại trừ, ta xét bài toán tìm thành phần cực đại trong một mảng các số thực. Ta sẽ thấy mọi thuật toán EREW cho bài toán này đều mất $\Omega(\lg n)$ thời gian và không có thuật toán CREW nào thực hiện tốt hơn. Bài toán có thể được giải trong $O(1)$ thời gian dùng một thuật toán CRCW chung, ở đó khi vài bộ xử lý ghi ra cùng vị trí, tất cả chúng đều ghi cùng giá trị.

Thuật toán CRCW tìm cực đại của n thành phần mảng mặc nhận rằng mảng đầu vào là $A[0..n-1]$. Thuật toán sử dụng n^2 bộ xử lý, với mỗi bộ xử lý so sánh $A[i]$ và $A[j]$ với vài i và j trong miền $0 \le i, j \le n-1$. Trên thực tế, thuật toán thực hiện một ma trận các phép so sánh, và do đó ta có thể xem mỗi trong số n bộ xử lý như là có không những một chỉ số một chiều trong PRAM, mà còn một chỉ số hai chiều (i, j).

```
FAST-MAX(A)
1   n ← length[A]
2   for i ← 0 to n − 1, song song
3        do m[i] ← TRUE
4   for i ← 0 to n − 1 và j ← 0 to n − 1, song song
5        do if A[i] < A[j]
6              then m[i] ← FALSE
7   for i ← 0 to n − 1, song song
8        do if m[i] = TRUE
9              then max ← A[i]
10  return max
```

Dòng 1 đơn giản xác định chiều dài của mảng A; nó chỉ cần được thi hành trên một bộ xử lý, giả sử bộ xử lý 0. Ta dùng một mảng $m[0..n-1]$,

ở đó bộ xử lý i chịu trách nhiệm $m[i]$. Ta muốn $m[i]$ = TRUE nếu và chỉ nếu $A[i]$ là giá trị cực đại trong mảng A. Ta bắt đầu (các dòng 2-3) bằng cách tin rằng mỗi thành phần mảng có thể là cực đại, và ta dựa trên các phép so sánh trong dòng 5 để xác định các thành phần mảng nào không phải là cực đại.

Hình 30.6 minh họa số dư của thuật toán. Trong vòng lặp của các dòng 4-6, ta kiểm tra mỗi cặp có thứ tự của các thành phần trong mảng A. Với mỗi cặp $A[i]$ và $A[j]$, dòng 5 kiểm tra xem $A[i] < A[j]$ hay không. Nếu so sánh này là TRUE, ta biết rằng $A[i]$ không thể là cực đại, và dòng 6 ấn định $m(i) \leftarrow$ FALSE để ghi nhận sự việc này. Vài cặp (i, j) có thể đang ghi ra $m[i]$ cùng một lúc, nhưng tất cả chúng đều ghi cùng giá trị: FALSE.

		$A[j]$					
		5	6	9	2	9	m
	5	F	T	T	F	T	F
	6	F	F	T	F	T	F
$A[i]$	9	F	F	F	F	F	T
	2	T	T	T	F	T	F
	9	F	F	F	F	F	T
						max	9

Hình 30.6 Tìm cực đại của n giá trị trong $O(1)$ thời gian bằng thuật toán CRCW FAST-MAX. Với mỗi cặp có sắp xếp của các thành phần trong mảng đầu vào A = <5, 6, 9, 2, 9>, kết quả của phép so sánh $A[i] < A[j]$ được nêu trong ma trận, được viết tắt là T thay cho TRUE và F thay cho FALSE. Với bất kỳ hàng nào chứa một giá trị TRUE, thành phần tương ứng của m, nêu ở bên phải, được ấn định là FALSE. Các thành phần của m chứa TRUE tương ứng với các thành phần có cực đại của A. Trong trường hợp này, giá trị 9 được ghi vào biến *max*.

Do đó, sau khi thi hành dòng 6, $m[i]$ = TRUE với chính xác các chỉ số i sao cho $A[i]$ đạt cực đại. Sau đó, các dòng 7-9 đưa giá trị cực đại vào biến *max*, được trả về trong dòng 10. Vài bộ xử lý có thể ghi vào biến *max*, nhưng nếu chúng thực hiện, tất cả chúng đều ghi cùng giá trị, nhất quán với mô hình PRAM CRCW chung.

Bởi cả ba "vòng lặp" trong thuật toán được thi hành song song, nên FAST-MAX chạy trong $O(1)$ thời gian. Tất nhiên, nó không hiệu quả công, bởi nó yêu cầu n^2 bộ xử lý, và bài toán tìm số cực đại trong một

mảng có thể được giải bằng một thuật toán nối tiếp $\Theta(n)$-thời gian. Tuy nhiên, ta có thể tiến gần hơn đến một thuật toán hiệu quả công, như Bài tập 30.2-6 yêu cầu bạn chứng tỏ.

Theo một nghĩa nào đó, điểm cốt lõi cho FAST-MAX đó là một PRAM CRCW có khả năng thực hiện một AND *bool n* biến trong $O(1)$ thời gian với n bộ xử lý. (Bởi khả năng này đứng vững trong mô hình CRCW chung, nên nó cũng đứng vững trong các mô hình PRAM CRCW mạnh hơn.) Mã thực tế thực hiện vài AND cùng một lúc , tính toán với i,= 0, 1,..., $n - 1$

$$m[i] = \bigwedge_{j=0}^{n-1} (A[i] \geq A[j]) ,$$

có thể được phái sinh từ luật DeMorgan (5.2). Khả năng AND mạnh này có thể được dùng theo các cách khác nhau. Ví dụ, khả năng của một PRAM CRCW để thực hiện một AND trong $O(1)$ thời gian loại bỏ nhu cầu để một mạng điều khiển tách biệt kiểm tra xem tất cả các bộ xử lý có hoàn tất việc lặp lại một vòng lặp hay không, như ta đã mặc nhận cho các thuật toán EREW. Quyết định hoàn tất vòng lặp đơn giản là AND của tất cả các lời đề nghị của các bộ xử lý muốn hoàn tất vòng lặp.

Mô hình EREW không có khả năng AND mạnh này. Mọi thành phần EREW tính toán cực đại của n thành phần đều chiếm $\Omega(\lg n)$ thời gian.

Về khái niệm, phần chứng minh tương tự như lập luận cận thấp hơn để tìm gốc của một cây nhị phân. Trong phần chứng minh đó, ta đã xem xét số lần các nút j có thể "biết" đồng nhất thức của gốc và đã chứng tỏ nó tối đa nhân đôi với mỗi bước. Với bài toán tính toán cực đại của n thành phần, ta xét số lượng các thành phần "nghĩ" rằng chúng có thể là cực đại. Theo trực giác, với mỗi bước của một PRAM EREW, con số này có thể tối đa chia đôi, dẫn đến $\Omega(\lg n)$ cận dưới.

Đáng lưu ý, $\Omega(\lg n)$ cận dưới để tính toán cực đại sẽ đứng vững cho dù ta cho phép đọc đồng thời; nghĩa là, nó đứng vững với các thuật toán CREW. Thực tế, Cook, Dwork, và Reischuk [50] chứng tỏ mọi thuật toán CREW để tìm cực đại của n thành phần phải chạy trong $\Omega(\lg n)$ thời gian, thậm chí với một số lượng bộ xử lý không hạn chế và bộ nhớ không hạn chế. Cận dưới của chúng cũng áp dụng cho bài toán tính

toán AND của n giá trị bool.

Mô phỏng một thuật toán CRCW với một thuật toán EREW

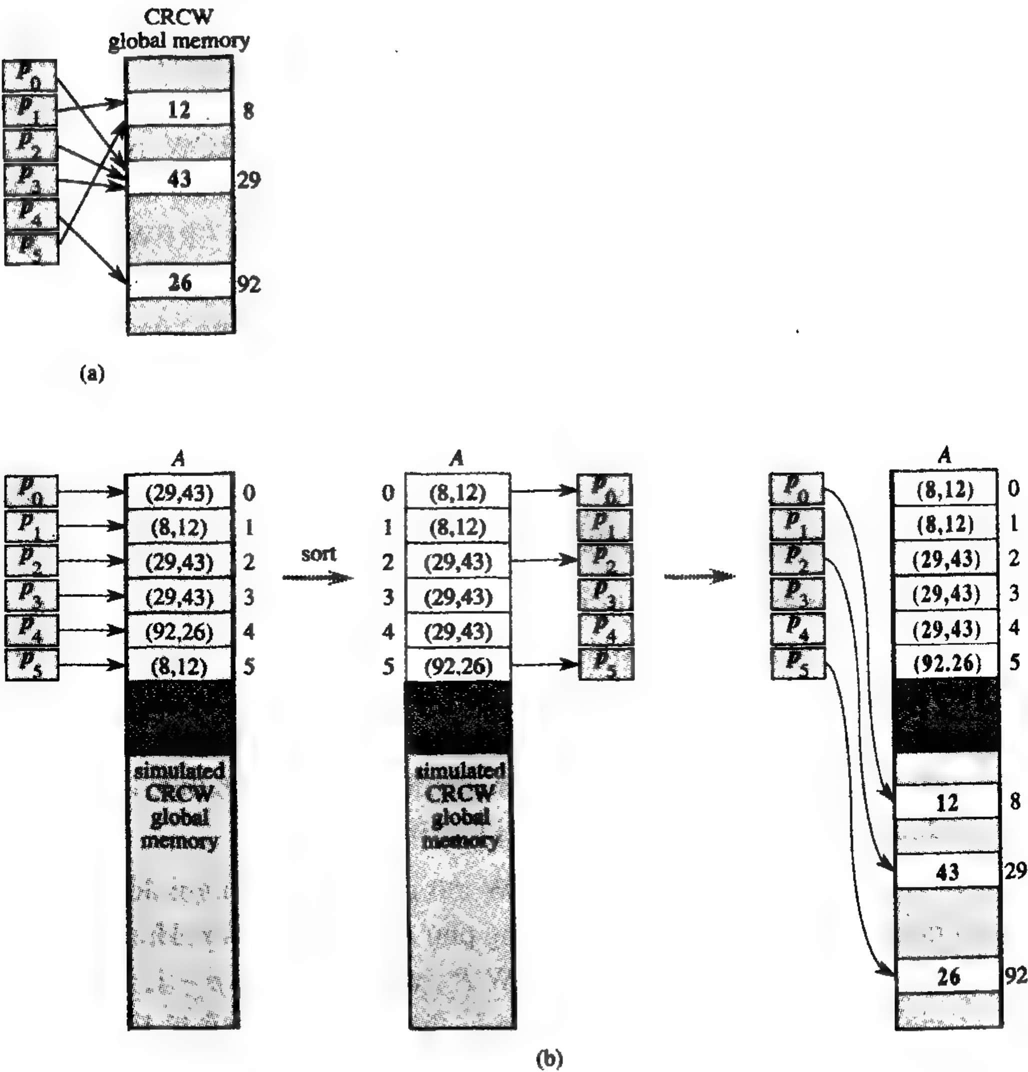
Giờ đây ta biết rằng các thuật toán CRCW có thể giải vài bài toán nhanh chóng hơn so với các thuật toán EREW. Hơn nữa, mọi thuật toán EREW đều có thể được thi hành trên một PRAM CRCW. Như vậy, mô hình CRCW hoàn toàn mạnh hơn mô hình EREW. Nhưng mạnh hơn bao nhiêu? Trong Đoạn 30.3, ta sẽ chứng tỏ một PRAM EREW p-bộ xử lý có thể sắp xếp p con số trong $O(\lg p)$ thời gian. Giờ đây ta dùng kết quả này để cung cấp một cận trên lý thuyết trên năng lực của một PRAM CRCW so với một PRAM EREW.

Định lý 30.1

Một thuật toán CRCW p-bộ xử lý có thể không hơn $O(\lg p)$ lần nhanh hơn thuật toán EREW p-bộ xử lý tốt nhất với cùng bài toán.

Chứng minh Phần chứng minh là một lập luận của phép mô phỏng. Ta mô phỏng mỗi bước của thuật toán CRCW với một phép tính EREW $O(\lg p)$-thời gian. Bởi năng lực xử lý của cả hai máy là như nhau, ta chỉ cần tập trung vào tiến trình truy cập bộ nhớ. Ở đây ta chỉ trình bày phần chứng minh để mô phỏng các lần ghi đồng thời. Phần thực thi tiến trình đọc đồng thời xin để lại làm Bài tập 30.2-8.

p bộ xử lý trong PRAM EREW mô phỏng một lần ghi đồng thời của thuật toán CRCW dùng một mảng phụ A có chiều dài p. Hình 30.7 minh họa ý tưởng. Khi bộ xử lý CRCW P_i, với $i = 0, 1, ..., p - 1$, mong muốn viết một dữ liệu x_i vào một vị trí l_i, thay vì mỗi bộ xử lý EREW tương ứng P_i ghi cặp có sắp xếp (l_i, x_i) ra vị trí $A[i]$. Các lần ghi này là loại trừ, bởi mỗi bộ xử lý ghi ra một vị trí bộ nhớ riêng biệt. Sau đó, mảng A được sắp xếp bởi tọa độ đầu tiên của các cặp có sắp xếp trong $O(\lg p)$ thời gian, khiến tất cả dữ liệu được ghi ra cùng vị trí được gom chung với nhau trong kết xuất.



Hình 30.7 Mô phỏng một lần ghi đồng thời trên một PRAM EREW. **(a)** Một bước của một thuật toán CRCW chung ở đó 6 bộ xử lý ghi đồng thời ra bộ nhớ toàn cục. **(b)** Mô phỏng bước trên một PRAM EREW. Trước tiên, các cặp có thứ tự chứa vị trí và dữ liệu được ghi ra một mảng *A*. Sau đó, mảng được sắp xếp. Bằng cách so sánh các thành phần kề trong mảng, ta bảo đảm chỉ lần ghi đầu tiên của mỗi nhóm các lần ghi giống nhau vào bộ nhớ toàn cục mới được thực thi. Trong trường hợp này, các bộ xử lý P_0, P_2, và P_5 thực hiện lần ghi.

Mỗi bộ xử lý EREW P_i, với $i = 1, 2,..., p - 1$, giờ đây thẩm tra $A[i] = (l_j, x_j)$ và $A[i - 1] = (l_k, x_k)$, ở đó j và k là những giá trị trong miền $0 \leq j, k \leq p - 1$. Nếu $l_j \neq l_k$ hoặc $i = 0$, thì bộ xử lý P_i, với $i = 0, 1,..., p - 1$, sẽ ghi dữ liệu x_j ra vị trí l_j trong bộ nhớ toàn cục. Bằng không, bộ xử lý không làm gì cả. Bởi mảng A được sắp xếp bởi tọa độ đầu tiên, chỉ một trong số các bộ xử lý ghi ra một vị trí bất kỳ đã cho mới thực tế thành công, và như vậy lần ghi là loại trừ. Do đó, tiến trình này thực thi mỗi bước của tiến trình ghi đồng thời trong mô hình CRCW chung trong $O(\lg p)$ thời gian.

Các mô hình khác dành cho tiến trình ghi đồng thời cũng có thể được mô phỏng. (Xem Bài tập 30.2-9.)

Do đó, vấn đề nảy sinh đó là mô hình nào được ưa chuộng—CRCW hay EREW—và nếu là CRCW, thì mô hình CRCW nào. Những người bênh vực cho các mô hình CROW chỉ ra rằng chúng dễ lập trình hơn mô hình EREW và các thuật toán của chúng chạy nhanh hơn. Những người phê bình lại cho rằng phần cứng để thực thi các phép toán bộ nhớ đồng thời chạy chậm hơn phần cứng để thực thi các phép toán bộ nhớ loại trừ, và như vậy thời gian thực hiện nhanh hơn của các thuật toán CRCW là hư cấu. Trong thực tế, chúng nói, ta không thể tìm ra cực đại của n giá trị trong $O(1)$ thời gian.

Những người khác nói rằng PRAM—EREW hay CRCW—là mô hình sai hoàn toàn. Các bộ xử lý phải được tương kết bằng một mạng truyền thông, và mạng truyền thông phải là một phần của mô hình. Các bộ xử lý chỉ có thể truyền thông với các láng giềng của chúng trong mạng.

Điều khá rõ ràng đó là vấn đề mô hình song song "đúng" ắt không dễ gì ổn định theo bất kỳ một mô hình nào. Tuy nhiên, điều quan trọng cần nhận thức đó là các mô hình này chỉ là : những mô hình. Với một tình huống đời thực đã cho, các mô hình khác nhau áp dụng các chừng mực khác nhau. Mức độ mà mô hình so khớp tình huống thiết kế kỹ thuật là mức độ mà các cuộc phân tích thuật toán trong mô hình sẽ tiên đoán các hiện tượng đời thực. Do đó, ta phải nghiên cứu các mô hình và các thuật toán song song khác nhau để khi lĩnh vực tính toán song song tăng trưởng, có thể sẽ nảy ra một sự đồng lòng sáng tỏ về những mô thức tính toán song song thích hợp nhất để thực thi.

Bài tập

30.2-1

Giả sử ta biết một rừng các cây nhị phân chỉ bao gồm một cây đơn với n nút. Chứng tỏ với giả thiết này, có thể thực hiện một thực thi CREW của FIND-ROOTS để chạy trong $O(1)$ thời gian, độc lập với độ

sâu của cây. Chứng tỏ mọi thuật toán EREW đều mất $\Omega(\lg n)$ thời gian.

30.2-2

Nêu một thuật toán EREW cho FIND-ROOTS chạy trong $O(\lg n)$ thời gian trên một rừng n nút.

30.2-3

Nếu một thuật toán CRCW n-bộ xử lý có thể tính toán OR của n giá trị bool trong $O(1)$ thời gian.

30.2-4

Mô tả một thuật toán CRCW hiệu quả để nhân hai ma trận bool $n \times n$ dùng n^3 bộ xử lý.

30.2-5

Mô tả một thuật toán EREW $O(\lg n)$-thời gian để nhân hai ma trận $n \times n$ gồm các số thực dùng n^3 bộ xử lý. Có thể có một thuật toán CRCW chung nhanh hơn không? Có thể có một thuật toán nhanh hơn theo một trong các mô hình CRCW mạnh hơn không?

30.2-6 *

Chứng minh với một hằng $\varepsilon > 0$, ta có một thuật toán CRCW $O(1)$-thời gian dùng $O(n^{1+\varepsilon})$ bộ xử lý để tìm thành phần cực đại của một mảng n thành phần.

30.2-7 *

Nêu cách trộn hai mảng đã sắp xếp, mỗi mảng với n số, trong $O(1)$ thời gian dùng một thuật toán CRCW ưu tiên. Mô tả cách sử dụng thuật toán này để sắp xếp trong $O(\lg n)$ thời gian. Thuật toán sắp xếp của bạn có hiệu quả công [work-efficient] không?

30.2-8

Hoàn tất Chứng minh của Định lý 30.1 bằng cách mô tả cách thực thi một lần đọc đồng thời trên một PRAM CRCW p-bộ xử lý trong $O(\lg p)$ thời gian trên một PRAM EREW p-bộ xử lý.

30.2-9

Nêu cách thức mà một PRAM EREW p-bộ xử lý có thể thực thi một PRAM CRCW phối hợp p-bộ xử lý với chỉ $O(\lg p)$ thất thoát khả năng thực hiện. (*Mách nước*: Dùng một phép tính tiền tố song song.)

30.3 Định lý Brent và tính hiệu quả công

Định lý Brent nêu cách mô phỏng hiệu quả một mạch tổ hợp bằng một PRAM. Dùng định lý này, ta có thể thích ứng nhiều kết quả của các mạng sắp xếp trong Chương 28 và nhiều kết quả của các mạch số học trong Chương 29 cho mô hình PRAM. Những bạn đọc chưa quen các mạch tổ hợp có thể nên xem lại Đoạn 29.1.

Một ***mạch tổ hợp*** là một mạng phi chu trình gồm ***các thành phần tổ hợp.*** Mỗi thành phần tổ hợp có một hoặc nhiều đầu vào, và trong đoạn này, ta sẽ mặc nhận rằng mỗi thành phần có chính xác một kết xuất. (Các thành phần tổ hợp với các kết xuất $k > 1$ có thể được xem là k thành phần tách biệt.) Số lượng đầu vào là ***mức lùa vào*** [fan-in] của thành phần, và số lượng vị trí mà kết xuất của nó nạp vào là ***mức lùa ra*** [fan-out] của nó. Nói chung, trong đoạn này ta mặc nhận rằng mọi thành phần tổ hợp trong mạch đã định cận mức lùa vào ($O(1)$). Tuy nhiên, nó có thể có mức lùa ra không định cận.

Kích cỡ của một mạch tổ hợp là số lượng các thành phần tổ hợp mà nó chứa. Số lượng các thành phần tổ hợp trên một lộ trình dài nhất từ một đầu vào của mạch đến một kết xuất của một thành phần tổ hợp chính là ***độ sâu*** của thành phần. ***Độ sâu*** của nguyên cả mạch là độ sâu cực đại của bất kỳ thành phần nào của nó.

Định lý 30.2 (Định lý Brent)

Bất kỳ mạch tổ hợp nào có độ sâu d, kích cỡ n với mức lùa vào được định cận đều có thể được mô phỏng bởi một thuật toán CREW p-bộ xử lý trong $O(n/p + d)$ thời gian.

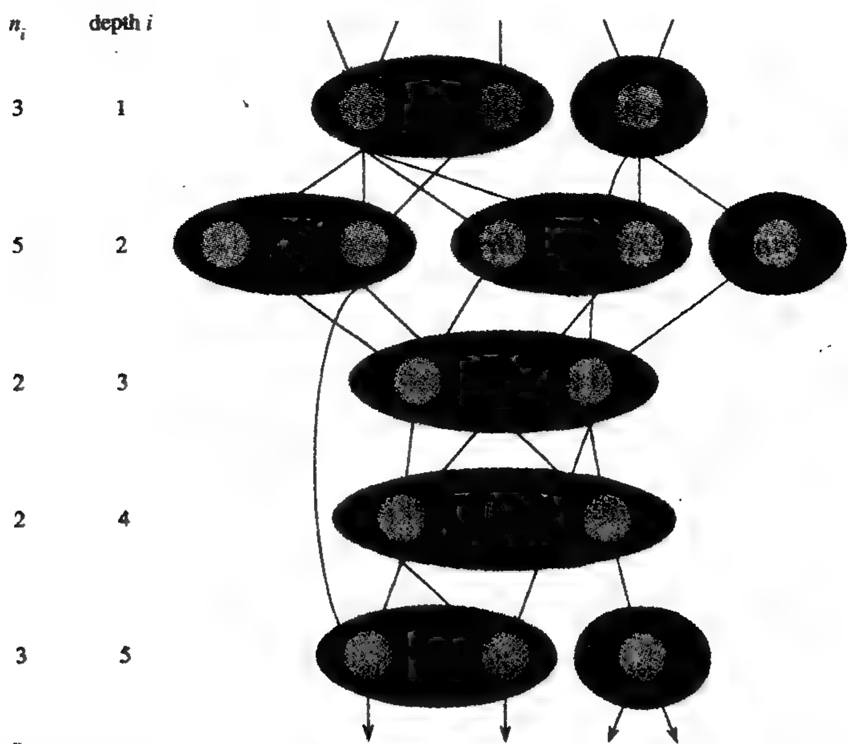
Chứng minh Ta lưu trữ các đầu vào đến mạch tổ hợp trong bộ nhớ toàn cục của PRAM, và ta dành riêng một vị trí cho mỗi thành phần tổ hợp trong mạch để lưu trữ giá trị kết xuất của nó khi nó được tính toán. Như vậy, một thành phần tổ hợp đã cho có thể được mô phỏng bởi một bộ xử lý PRAM đơn lẻ trong $O(1)$ thời gian như sau. Bộ xử lý đơn giản đọc các giá trị đầu vào cho thành phần từ các giá trị trong bộ nhớ tương ứng với các đầu vào mạch hoặc các kết xuất thành phần nạp nó, và do đó mô phỏng các dây dẫn trong mạch. Sau đó, nó tính toán hàm của thành phần tổ hợp và ghi kết quả vào vị trí thích hợp trong bộ nhớ. Bởi mức lùa vào của từng thành phần mạch được định cận, nên mỗi hàm có thể được tính toán trong $O(1)$ thời gian.

Do đó, công việc của chúng ta đó là tìm ra một lịch biểu cho p bộ xử lý của PRAM sao cho tất cả các thành phần tổ hợp được mô phỏng trong $O(n/p+d)$ thời gian. Sự ràng buộc chính đó là một thành phần không thể

được mô phỏng cho đến khi tính toán xong các kết xuất từ bất kỳ thành phần nào nạp nó. Các lần đọc đồng thời được sử dụng mỗi khi có vài thành phần tổ hợp được mô phỏng song song yêu cầu cùng giá trị.

Bởi tất cả các thành phần ở độ sâu 1 chỉ tùy thuộc vào các đầu vào mạch, nên chúng là những thành phần duy nhất có thể được mô phỏng trước tiên. Một khi chúng đã được mô phỏng, tất cả các thành phần ở độ sâu 2 có thể được mô phỏng, và vân vân, cho đến khi ta hoàn tất với tất cả các thành phần ở độ sâu d. Ý tưởng chính đó là nếu tất cả các thành phần từ các độ sâu 1 đến i đã được mô phỏng, ta có thể mô phỏng bất kỳ tập hợp con nào của các thành phần ở độ sâu $i + 1$ song song, bởi các phép tính của chúng độc lập nhau.

Do đó chiến lược lên lịch của chúng ta khá đơn sơ. Ta mô phỏng tất cả các thành phần tại độ sâu i trước khi tiếp tục mô phỏng những thành phần ở độ sâu $i + 1$. Trong một độ sâu đã cho i, ta mô phỏng các thành phần p cùng một lúc. Hình 30.8 minh họa một chiến lược như vậy cho $p = 2$.



Hình 30.8 Định lý Brent. Mạch tổ hợp có kích cỡ 15 và độ sâu 5 được mô phỏng bởi một PRAM CREW 2 bộ xử lý trong $9 \le 15/2 + 5$ bước. Phép mô phỏng tiến hành từ trên xuống qua mạch. Các nhóm tô bóng của các thành phần mạch nêu rõ những thành phần được mô phỏng cùng một lúc, và mỗi nhóm được gán nhãn bằng một con số tương ứng với bước thời gian khi các thành phần của nó được mô phỏng.

Ta hãy phân tích chiến lược mô phỏng này. Với $i = 1, 2,..., d$, cho n_i là số lượng các thành phần ở độ sâu i trong mạch. Như vậy,

$$\sum_{i=1}^{d} n_i = n .$$

Xét các thành phần tổ hợp n_i ở độ sâu i. Bằng cách gom nhóm chúng vào $\lceil n_i / p \rceil$ nhóm, ở đó $\lfloor n_i /p \rfloor$ nhóm đầu tiên có p thành phần cho mỗi nhóm và các thành phần chưa giải, nếu có, sẽ nằm trong nhóm cuối, PRAM có thể mô phỏng các phép tính được thực hiện bởi các thành phần tổ hợp này trong $O(\lceil n_i /p \rceil)$ thời gian. Do đó, tổng thời gian mô phỏng nằm trên thứ tự của

$$\begin{aligned} \sum_{i=1}^{d} \left\lceil \frac{n_i}{p} \right\rceil &\le \sum_{i=1}^{d} \left(\frac{n_i}{p} + 1 \right) \\ &= \frac{n}{p} + d . \end{aligned}$$

Có thể mở rộng định lý Brent cho các phép mô phỏng EREW khi một mạch tổ hợp có mức lùa ra $O(1)$ cho mỗi thành phần tổ hợp.

Hệ luận 30.3

Bất kỳ mạch tổ hợp nào có độ sâu d, kích cỡ n với mức lùa vào được định cận và mức lùa ra có thể được mô phỏng trên một PRAM EREW p-bộ xử lý trong $O(n/p + d)$ thời gian.

Chứng minh Ta thực hiện một phép mô phỏng tương tự như trong phần chứng minh của định lý Brent. Sự khác biệt duy nhất nằm trong phép mô phỏng của các dây dẫn, là nơi Định lý 30.2 yêu cầu tính năng đọc đồng thời. Với phép mô phỏng EREW, sau khi kết xuất của một thành phần tổ hợp được tính toán, nó không được đọc trực tiếp bởi các bộ xử lý yêu cầu giá trị của nó. Thay vì thế, giá trị kết xuất được bộ xử lý mô.phỏng thành phần đó chép ra các đầu vào $O(1)$ yêu cầu nó. Sau đó, các bộ xử lý cần giá trị có thể đọc nó mà không cản trở nhau.

Chiến lược mô phỏng EREW này không làm việc cho các thành phần có mức lùa ra không định cận, bởi tiến trình chép có thể kéo dài hơn thời gian bất biến tại mỗi bước. Như vậy, với các mạch có các thành phần có mức lùa ra không định cận, ta cần năng lực của các lần đọc đồng thời. (Đôi lúc, ta có thể dùng phép mô phỏng CRCW để điều quản trường hợp của mức lùa vào không định cận nếu các thành phần tổ hợp đơn giản là đủ. Xem Bài tập 30.3-1.)

Hệ luận 30.3 cung cấp một thuật toán sắp xếp EREW nhanh. Như đã giải thích trong phần ghi chú chương của Chương 28, mạng sắp xếp AKS có thể sắp xếp n số trong độ sâu $O(\lg n)$ dùng $O(n \lg n)$ bộ so sánh.

Do các bộ so sánh đã định cận mức lùa vào, nên ta có một thuật toán EREW để sắp xếp n số trong $O(\lg n)$ thời gian dùng n bộ xử lý. (Ta đã dùng kết quả này trong Định lý 30.1 để chứng tỏ một PRAM EREW có thể mô phỏng một PRAM CRCW với tối đa sự giảm tốc loga.) Đáng tiếc, các hằng mà hệ ký hiệu O che giấu lại quá lớn đến nỗi thuật toán sắp xếp này chỉ đáng quan tâm về mặt lý thuyết. Tuy nhiên, người ta đã khám phá các thuật toán sắp xếp EREW mang tính thực tiễn hơn, đáng chú ý là thuật toán sắp xếp trộn song song của Cole [46].

Giờ đây giả sử rằng ta có một thuật toán PRAM sử dụng tối đa p bộ xử lý, nhưng ta có một PRAM với chỉ $p' < p$ bộ xử lý. Ta muốn có khả năng chạy thuật toán p-bộ xử lý trên PRAM p'-bộ xử lý nhỏ hơn theo cách hiệu quả công. Nhờ dùng ý tưởng trong phần chứng minh của định lý Brent, ta có thể gán một điều kiện khi điều này khả dĩ.

Định lý 30.4

Nếu một thuật toán PRAM p-bộ xử lý A chạy trong thời gian t, thì với bất kỳ $p' < p$, ta có một thuật toán PRAM p'-bộ xử lý A' cho cùng bài toán chạy trong thời gian $O(pt/p')$.

Chứng minh Cho các bước thời gian của thuật toán A được đánh số 1, 2,..., t. Thuật toán A' mô phỏng cách thi hành của mỗi lần bước i = 1, 2,..., t trong thời gian $O(\lceil p/p' \rceil)$. Có t bước, và do đó nguyên cả phép mô phỏng mất một thời gian $O(\lceil p/p' \rceil t) = O(pt/p')$, bởi $p' < p$.

Công mà thuật toán A thực hiện là pt, và công mà thuật toán A' thực hiện là $(pt/p')p' = pt$; do đó phép mô phỏng là hiệu quả công. Bởi vậy, nếu bản thân thuật toán A là hiệu quả công, thì thuật toán A' cũng vậy.

Do đó, khi phát triển các thuật toán hiệu quả công cho một bài toán, ta chẳng cần nhất thiết tạo một thuật toán khác nhau cho từng số lượng bộ xử lý khác nhau. Ví dụ, giả sử ta có thể chứng minh một cận dưới chặt của t trên thời gian thực hiện của bất kỳ thuật toán song song nào, bất kể là bao nhiêu bộ xử lý, để giải một bài toán đã cho, và giả sử thêm rằng thuật toán nối tiếp tốt nhất cho bài toán sẽ thực hiện công w. Như vậy, ta chỉ cần phát triển một thuật toán hiệu quả công cho bài toán sử dụng $p = \Theta(w/t)$ bộ xử lý để có được các thuật toán hiệu quả công với tất cả các số lượng bộ xử lý khả dĩ cho một thuật toán hiệu quả công. Với $p' = o(p)$, Định lý 30.4 bảo đảm có một thuật toán hiệu quả công. Với $p' = \omega(p)$ không có thuật toán hiệu quả công nào tồn tại, bởi nếu t là một cận dưới trên thời gian với một thuật toán song song bất kỳ, $p't = \omega(pt) = \omega(w)$.

Bài tập

30.3-1

Chứng minh một kết quả tương tự như định lý Brent cho một phép mô phỏng CRCW của các mạch tổ hợp bool có các cổng AND và OR với mức lùa vào không định cận. (*Mách nước*: Cho "kích cỡ" là tổng của các đầu vào đến các cổng trong mạch.)

30.3-2

Chứng tỏ một phép tính tiền tố song song trên n giá trị được lưu trữ trong một mảng của bộ nhớ có thể được thực thi trong $O(\lg n)$ thời gian trên một PRAM EREW dùng $O(n/\lg n)$ bộ xử lý. Tại sao kết quả này không mở rộng tức thời đến một danh sách n giá trị?

30.3-3

Nêu cách nhân một ma trận $n \times n$ A với một b n-vector trong $O(\lg n)$ thời gian bằng một thuật toán EREW hiệu quả công. (*Mách nước*: Kiến tạo một mạch tổ hợp cho bài toán.)

30.3-4

Nêu một thuật toán CROW dùng n^2 bộ xử lý để nhân hai ma trận $n \times n$. Thuật toán phải là hiệu quả công đối với thuật toán nối tiếp $\Theta(n^3)$ thời gian bình thường để nhân các ma trận. Có thể tạo thuật toán EREW không?

30.3-5

Vài mô hình song song cho phép các bộ xử lý trở thành không hoạt động, sao cho số lượng bộ xử lý thi hành tại một bước bất kỳ sẽ thay đổi. Định nghĩa công [work] trong mô hình này dưới dạng tổng các bước được thi hành trong một thuật toán bằng các bộ xử lý hoạt động. Chứng tỏ mọi thuật toán CRCW thực hiện công w và chạy trong t thời gian có thể chạy trên một PRAM EREW p-bộ xử lý trong $O((w/p + t) \lg p)$ thời gian. (*Mách nước*: Phần khó đó là lên lịch các bộ xử lý hoạt động *trong khi* phép tính đang tiến hành.)

---

* 30.4 Phép tính tiền tố song song hiệu quả công

Trong Đoạn 30.1.2, ta đã xem xét một thuật toán EREW LIST-RANK $O(\lg n)$-thời gian có thể thực hiện một phép tính tiền tố trên một danh sách nối kết n-đối tượng. Thuật toán sử dụng n bộ xử lý và thực hiện $\Theta(n \lg n)$ công. Bởi ta có thể dễ dàng thực hiện một phép tính tiền tố

trong $\Theta(n)$ thời gian trên một máy nối tiếp, nên LIST-RANK không hiệu quả công.

Đoạn này trình bày một thuật toán tiền tố song song ngẫu nhiên hóa EREW là hiệu quả công. Thuật toán sử dụng $\Theta(n/\lg n)$ bộ xử lý, và chạy trong $O(\lg n)$ thời gian với xác suất cao. Như vậy, nó là hiệu quả công với xác suất cao. Hơn nữa, theo Định lý 30.4, thuật toán này lập tức cho ra các thuật toán hiệu quả công với một số $p = O(n/\lg n)$ bộ xử lý.

Phép tính tiền tố song song đệ quy

Thuật toán tiền tố song song ngẫu nhiên hóa RANDOMIZED-LIST-PREFIX hoạt động trên một danh sách nối kết n đối tượng dùng $p = \Theta(n/\lg n)$ bộ xử lý. Trong khi thi hành thuật toán, mỗi bộ xử lý chịu trách nhiệm $n/p = \Theta(\lg n)$ đối tượng trong danh sách ban đầu. Các đối tượng được gán cho các bộ xử lý một cách tùy ý (không nhất thiết liền kề) trước khi đệ quy bắt đầu, và "quyền sở hữu" của các đối tượng không bao giờ thay đổi. Để tiện dụng, ta mặc nhận rằng danh sách được nối kết gấp đôi, bởi việc nối kết gấp đôi chiếm $O(1)$ thời gian.

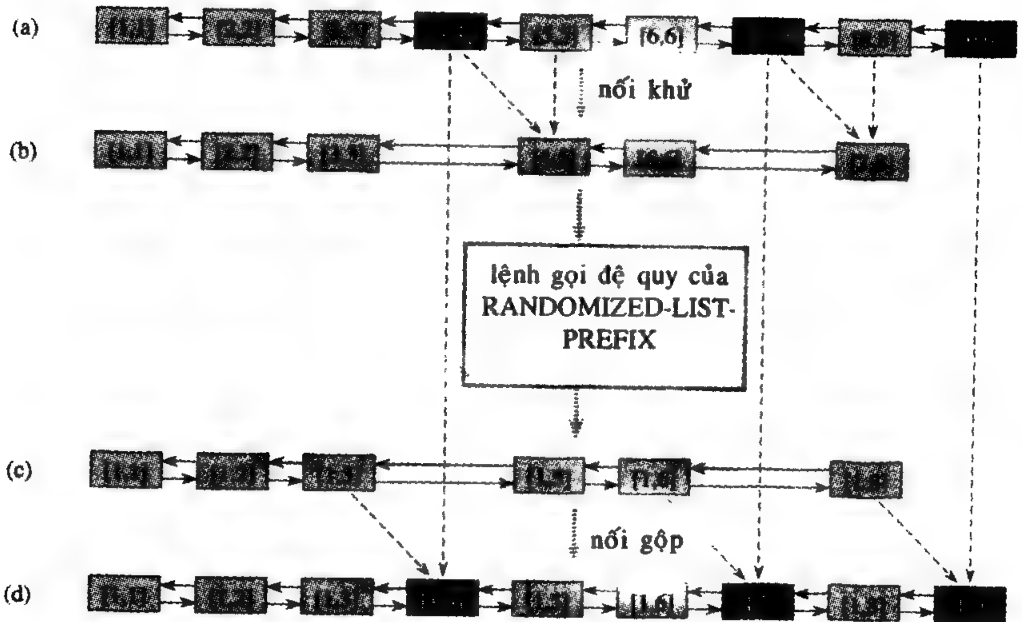
Ý tưởng của RANDOMIZED-LIST-PREFIX đó là loại bỏ vài đối tượng trong danh sách, thực hiện một phép tính tiền tố đệ quy trên danh sách kết quả, rồi mở rộng nó bằng cách nối gộp [splicing in] các đối tượng đã loại để cho ra một phép tính tiền tố trên danh sách ban đầu. Hình 30.9 minh họa tiến trình đệ quy, và Hình 30.10 nêu cách trải rộng đệ quy. Dưới đây ta sẽ chứng tỏ khái quát rằng mỗi giai đoạn của đệ quy tuân thủ hai tính chất:

1. Tối đa một đối tượng trong số thuộc về một bộ xử lý đã cho sẽ được chọn để loại bỏ.

2. Không có hai đối tượng kề nhau được chọn để loại bỏ.

Trước khi nêu cách chọn các đối tượng thỏa các tính chất này, ta hãy xét kỹ hơn cách thực hiện phép tính tiền tố. Giả sử rằng tại bước đầu tiên của đệ quy, đối tượng thứ k trong danh sách được chọn để loại bỏ. Đối tượng này chứa giá trị $[k, k]$, được truy nạp bởi đối tượng thứ $(k + 1)$ trong danh sách. (Các tình huống biên, như trường hợp ở đây khi k ở cuối danh sách, có thể được điều quản một cách dễ dàng và không được mô tả.) Đối tượng thứ $(k + 1)$, lưu giữ giá trị $[k + 1, k + 1]$, sau đó tính toán và lưu trữ $[k, k + 1] = [k, k] \otimes [k + 1, k + 1]$. Vậy, đối tượng thứ k được loại bỏ ra khỏi danh sách bằng cách nối khử nó.

Sau đó, thủ tục RANDOMIZED-LIST-PREFIX tự gọi một cách đệ quy để thực hiện một phép tính tiền tố trên danh sách "cô đọng". (Đệ quy xuống mức thấp nhất khi nguyên cả danh sách trống rỗng.) Nhận

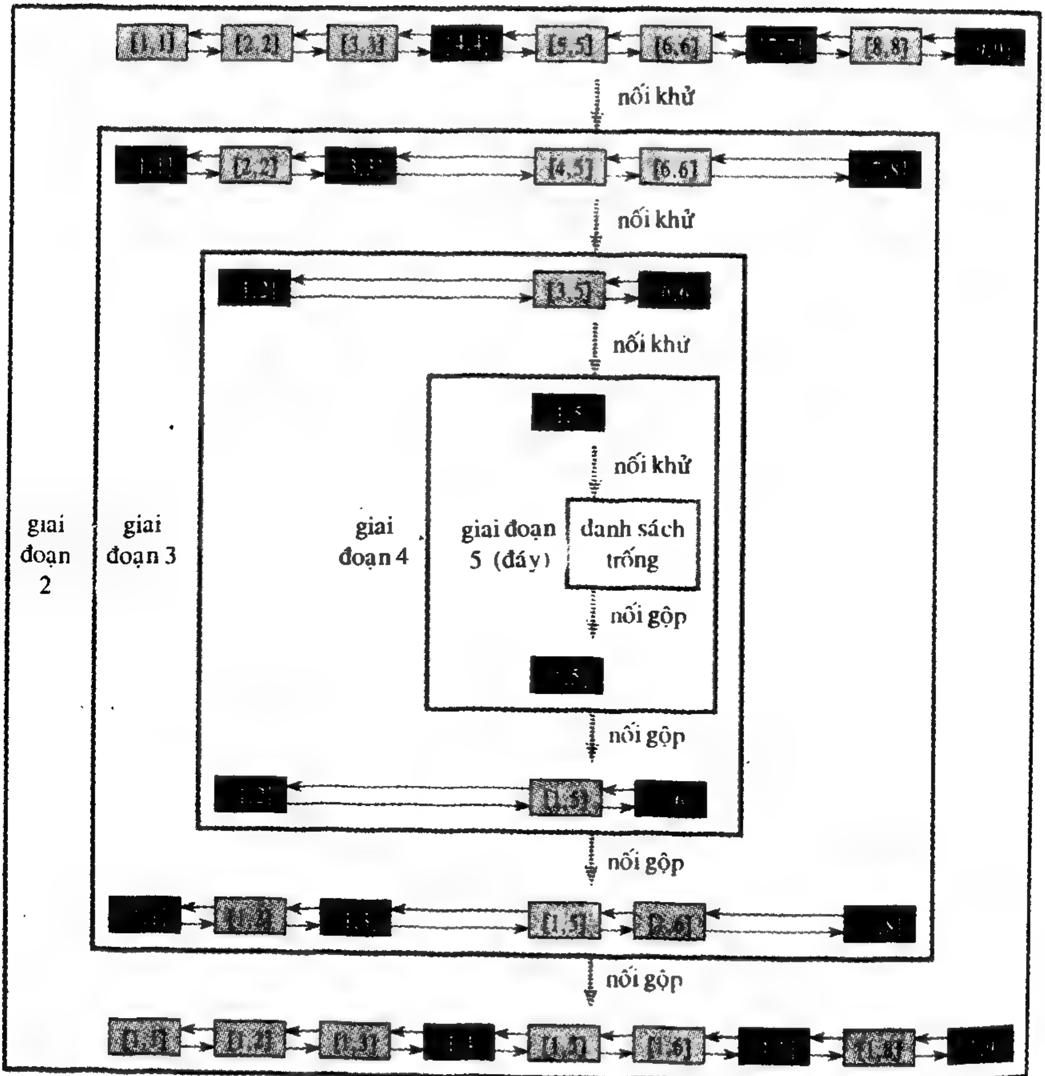


Hình 30.9 Thuật toán song song hiệu quả công, ngẫu nhiên hóa, đệ quy, RANDOMIZED-LIST-PREFIX, để thực hiện các phép tính tiền tố trên một danh sách nối kết $n = 9$ đối tượng. **(a)-(b)** Một tập hợp các đối tượng không liền kề (tô đen) được chọn để loại trừ. Giá trị trong mỗi đối tượng đen được dùng để cập nhật giá trị trong đối tượng kế tiếp trong danh sách, rồi đối tượng đen được nối khử [splice out]. Thuật toán được gọi theo đệ quy để tính toán một tiền tố song song trên danh sách cô đọng. **(c)-(d)** Các giá trị kết quả là các giá trị đúng đắn chung cuộc cho các đối tượng trong danh sách cô đọng. Sau đó, các đối tượng đã loại được nối gộp trở lại, và mỗi đối tượng sử dụng giá trị của đối tượng trước đó để tính toán giá trị chung cuộc của nó.

xét chính đó là sau khi trở về từ lệnh gọi đệ quy, mỗi đối tượng trong danh sách cô đọng có giá trị chung cuộc đúng đắn mà nó cần cho phép tính tiền tố song song trên danh sách ban đầu. Ta chỉ việc nối gộp trở lại các đối tượng đã loại trước đó, như đối tượng thứ k, và cập nhật các giá trị của chúng.

Sau khi đối tượng thứ k được nối gộp, giá trị tiền tố chung cuộc củanó có thể được tính toán nhờ dùng giá trị trong đối tượng thứ $(k - 1)$. Sau khi đệ quy, đối tượng thứ $(k - 1)$ chứa $[1, k - 1]$, và như vậy đối tượng thứ k—vẫn có giá trị $[k, k]$—chỉ cần truy nạp giá trị $[1, k - 1]$ và tính toán $[1, k] = [1, k - 1] \otimes [k, k]$.

Bởi tính chất 1, nên mỗi đối tượng đã chọn sẽ có một bộ xử lý riêng biệt để thực hiện công cần có để nối khử hoặc nối gộp nó. Tính chất 2 bảo đảm không có lẫn lộn giữa các bộ xử lý nảy sinh khi nối khử và nối gộp các đối tượng (xem Bài tập 30.4-1). Hai tính chất chung nhau bảo đảm mỗi bước đệ quy có thể được thực thi trong $O(1)$ thời gian theo kiểu EREW.



Hình 30.10 Các giai đoạn đệ quy của RANDOMIZED-LIST-PREFIX, được nêu với $n = 9$ đối tượng ban đầu. Trong mỗi giai đoạn, các đối tượng tô đen được loại bỏ. Thủ tục sẽ đệ quy cho đến khi danh sách trống rỗng, sau đó các đối tượng đã loại được nối gộp.

Chọn các đối tượng để loại bỏ

RANDOMIZED-LIST-PREFIX chọn các đối tượng để loại bỏ ra sao? Nó phải tuân thủ hai tính chất nêu trên, và trong phép cộng, ta muốn thời gian để lựa các đối tượng phải ngắn (và tốt nhất là không đổi). Hơn nữa, ta muốn chọn càng nhiều đối tượng càng tốt.

Phương pháp dưới đây để chọn ngẫu nhiên thỏa các điều kiện này. Các đối tượng được chọn bằng cách để mỗi bộ xử lý thi hành các bước sau đây:

1. Bộ xử lý chọn một đối tượng *i* chưa được lựa trước đó trong số một nó sở hữu.

2. Sau đó, nó "lật một đồng xu," chọn các giá trị HEAD và TAIL theo xác suất bằng nhau.

3. Nếu nó chọn HEAD, nó đánh dấu đối tượng *i* là đã chọn, trừ phi *next*[*i*] đã được chọn bởi một bộ xử lý khác cũng có đồng xu là HEAD.

Phương pháp ngẫu nhiên hóa nảy chỉ mất $O(1)$ thời gian để lựa các đối tượng để loại bỏ, và nó không yêu cầu các lần truy cập bộ nhớ đồng thời.

Ta phải chứng tỏ thủ tục này tuân thủ hai tính chất trên đây. Ta có thể dễ dàng thấy tính chất 1 đứng vững, bởi mỗi bộ xử lý chỉ có thể chọn một đối tượng. Để xem tính chất 2 đứng vững, ta giả sử ngược lại rằng hai đối tượng liên tiếp *i* và *next*[*i*] được chọn. Điều này chỉ xảy ra nếu cả hai đã được chọn bởi các bộ xử lý của chúng, và cả hai bộ xử lý đã lật HEAD. Nhưng đối tượng *i* không được chọn nếu bộ xử lý chịu trách nhiệm *next*[*i*] đã lật HEAD, là một mâu thuẫn.

Phân tích

Bởi mỗi bước đệ quy của RANDOMIZED-LIST-PREFIX chạy trong $O(1)$ thời gian, nên để phân tích thuật toán ta chỉ cần xác định nó thực hiện bao nhiêu bước nó để loại bỏ tất cả các đối tượng trong danh sách ban đầu. Tại mỗi bước, một bộ xử lý có ít nhất xác suất 1/4 để loại bỏ đối tượng *i* mà nó chọn. Tại sao? Nó lật HEAD với xác suất 1/2, và xác suất mà nó không chọn *next*[*i*] hoặc chọn nó và lật TAIL ít nhất là 1/2. Bởi hai lần tung đồng xu là các sự kiện độc lập, nên ta có thể nhân các xác suất của chúng, cho ra xác suất ít nhất 1/4 để một bộ xử lý loại bỏ đối tượng mà nó nhận. Bởi mỗi bộ xử lý sở hữu $\Theta(\lg n)$ đối tượng, nên thời gian dự trù để một bộ xử lý loại bỏ tất cả các đối tượng của nó sẽ là $\Theta(\lg n)$.

Đáng tiếc, kiểu phân tích đơn giản không chứng tỏ thời gian thực hiện dự trù của RANDOMIZED-LIST-PREFIX là $\Theta(\lg n)$. Ví dụ, nếu

hầu hết các bộ xử lý nhanh chóng loại bỏ tất cả các đối tượng của chúng và một số bộ xử lý kéo dài hơn nhiều, thời gian trung bình để một bộ xử lý loại bỏ tất cả các đối tượng của nó có thể vẫn là $\Theta(\lg n)$, nhưng thời gian thực hiện của thuật toán có thể lớn.

Thời gian thực hiện dự trù của thủ tục RANDOMIZED-LIST-PREFIX quả thực là $\Theta(\lg n)$, cho dù đợt phân tích đơn giản không nêu nó. Ta sẽ dùng một lập luận xác suất cao để chứng minh rằng với xác suất ít nhất $1 - 1/n$, tất cả đối tượng được loại bỏ trong $c \lg n$ giai đoạn của đệ quy, với một hằng c. Bài tập 30.4-4 và 30.4-5 yêu cầu bạn tổng quát hóa lập luận này để chứng minh cận $\Theta(\lg n)$ trên thời gian thực hiện dự trù.

Đối số xác suất cao dựa trên nhận xét rằng thí nghiệm của một bộ xử lý đã cho loại bỏ các đối tượng mà nó chọn có thể được xem như một dãy các lần thử Bernoulli (xem Chương 6). Thí nghiệm là một thành công nếu đối tượng được lựa để loại bỏ, và bằng không nó là một thất bại. Bởi ta quan tâm đến việc chứng tỏ xác suất để đạt được các lần thành công rất ít là nhỏ, nên ta có thể mặc nhận rằng các lần thành công xảy ra với xác suất chính xác 1/4, thay vì với xác suất ít nhất 1/4. (Xem Bài tập 6.4-8 và 6.4-9 để có phần xác minh chính thức của giả thiết tương tự.)

Để rút gọn hơn nữa phần phân tích, ta mặc nhận rằng có chính xác $n/\lg n$ bộ xử lý, mỗi bộ xử lý có $\lg n$ đối tượng danh sách. Ta sẽ tiến hành $c \lg n$ lần thử, với một hằng c mà ta sẽ xác định, và ta quan tâm đến sự kiện có ít hơn $\lg n$ lần thành công xảy ra. Cho X là biến ngẫu nhiên thể hiện tổng số lần thành công. Theo Hệ luận 6.3, xác suất mà một bộ xử lý loại bỏ ít hơn $\lg n$ đối tượng trong $c \lg n$ lần thử tối đa là

$$
\begin{aligned}
\Pr\{X < \lg n\} &\le \binom{c \lg n}{\lg n}\left(\frac{3}{4}\right)^{c\lg n - \lg n} \\
&\le \left(\frac{ec \lg n}{\lg n}\right)^{\lg n}\left(\frac{3}{4}\right)^{(c-1)\lg n} \\
&= \left(ec\left(\frac{3}{4}\right)^{c-1}\right)^{\lg n} \\
&\le \left(\frac{3}{4}\right)^{\lg n} \\
&= 1/n^2 ,
\end{aligned}
$$

miễn là $c \ge 20$. (Dòng thứ hai là do bất đẳng thức (6.9).) Như vậy, xác suất mà tất cả các đối tượng thuộc về một bộ xử lý đã cho chưa

được loại bỏ sau khi $c \lg n$ bước tối đa là $1/n^2$.

Giờ đây, ta muốn định cận xác suất mà tất cả các đối tượng thuộc về tất cả các bộ xử lý chưa được loại bỏ sau $c \lg n$ bước. Theo bất đẳng thức Boole (6.22), xác suất này tối đa là tổng các xác suất mà mỗi trong số các bộ xử lý chưa loại bỏ các đối tượng của nó. Bởi có $n/\lg n$ bộ xử lý, và mỗi bộ có xác suất tối đa $1/n^2$ không loại bỏ tất cả các đối tượng của nó, nên xác suất mà một bộ xử lý bất kỳ chưa hoàn tất tất cả các đối tượng của nó tối đa là

$$\frac{n}{\lg n} \cdot \frac{1}{n^2} \leq \frac{1}{n} .$$

Như vậy, ta đã chứng minh rằng với xác suất ít nhất $1 - 1/n$, mọi đối tượng được nối khử sau $O(\lg n)$ lệnh gọi đệ quy. Bởi mỗi lệnh gọi đệ quy chiếm $O(1)$ thời gian, nên RANDOMIZED-LIST-PREFIX kéo dài $O(\lg n)$ thời gian với xác suất cao.

Hằng $c \geq 20$ trong $c \lg n$ thời gian thực hiện có vẻ như hơi lớn đối với tính thực tiễn. Thực vậy, hàng này giống như một dồ tạo tác của phần phân tích hơn là một sự phản ánh khả năng thực hiện của thuật toán. Trong thực tế, thuật toán có khuynh hướng chạy nhanh. Các thừa số bất biến trong phân tích là lớn bởi sự kiện mà một bộ xử lý hoàn tất loại bỏ tất cả các đối tượng danh sách của nó sẽ lệ thuộc vào sự kiện mà một bộ xử lý khác hoàn tất toàn bộ công của nó. Bởi các yếu tố này, nên ta đã dùng bất đẳng thức Boole, không yêu cầu sự độc lập nhưng dẫn đến một hằng yếu hơn so với thường gặp trong thực tế.

Bài tập

30.4-1

Vẽ các hình minh họa nội dung có thể hóa sai trong RANDOMIZED-LIST-PREFIX nếu hai đối tượng danh sách kề nhau được lựa để loại bỏ.

30.4-2 *

Đề xuất một thay đổi đơn giản để RANDOMIZED-LIST-PREFIX chạy trong $O(n)$ thời gian trường hợp xấu nhất trên một danh sách n đối tượng. Dùng phần định nghĩa về giá trị kỳ vọng để chứng minh rằng với kiểu sửa đổi này, thuật toán chạy trong $O(\lg n)$ thời gian dự trù.

30.4-3 *

Nêu cách thực thi RANDOMIZED-LIST-PREFIX để nó sử dụng tối đa $O(n/p)$ không gian cho mỗi bộ xử lý trong trường hợp xấu nhất, độc

lập với độ sâu mà đệ quy tiến tới.

30.4-4 *

Chứng tỏ với một hằng $k \geq 1$, RANDOMIZED-LIST-PREFIX chạy trong $O(\lg n)$ thời gian với xác suất ít nhất $1 - 1/n^k$. Cho biết k ảnh hưởng như thế nào đến hằng trong cận thời gian thực hiện.

30.4-5

Dùng kết quả của Bài tập 30.4-4, chứng tỏ thời gian thực hiện dự trù của RANDOMIZED-LIST-PREFIX là $O(\lg n)$.

30.5 Ngắt tính đối xứng tất định

Xét tình huống ở đó hai bộ xử lý muốn thủ đắc quyền truy cập loại trừ lẫn nhau đối với một đối tượng. Làm sao các bộ xử lý có thể xác định cái nào sẽ thủ đắc quyền truy cập trước? Ta muốn tránh bối cảnh ở đó cả hai được giao quyền truy cập, cũng như bối cảnh ở đó không có cái nào được giao quyền truy cập. Bài toán chọn một trong các bộ xử lý là một ví dụ về ***ngắt tính đối xứng*** [symmetry breaking]. Tất cả chúng ta đã thấy sự lẫn lộn nhất thời và các thế bế tắc ngoại giao nảy sinh khi hai người cùng lúc gắng đi qua một cánh cửa. Các bài toán ngắt tính đối xứng tương tự rất phổ biến trong khi thiết kế các thuật toán song song, và các giải pháp hiệu quả là cực kỳ hữu ích.

Một phương pháp để ngắt tính đối xứng đó là tung các đồng xu. Trên một máy tính, việc tung đồng xu có thể được thực thi thông qua một bộ sinh ngẫu số. Với ví dụ hai bộ xử lý, cả hai bộ xử lý có thể tung các đồng xu. Nếu một cái được HEAD và cái kia được TAIL, cái có HEAD sẽ tiếp tục. Nếu cả hai đều lật cùng giá trị, chúng sẽ thử lại. Với chiến lược này, tính đối xứng bị ngắt trong thời gian dự trù không đổi (xem Bài tập 30.5-1).

Ta đã thấy tính hiệu quả của một chiến lược ngẫu nhiên hóa trong Đoạn 30.4. Trong RANDOMIZED-LIST-PREFIX, các đối tượng danh sách kề nhau không được lựa để loại bỏ, nhưng phải lựa càng nhiều đối tượng được chọn càng tốt. Tuy nhiên, ở giữa một danh sách các đối tượng đã chọn, tất cả các đối tượng trông có vẻ khá giống nhau. Như đã thấy, ngẫu nhiên hóa sẽ cung cấp một cách đơn giản và hiệu quả để ngắt tính đối xứng giữa các đối tượng danh sách kề nhau trong khi bảo đảm rằng, với xác suất cao, nhiều đối tượng được lựa.

Trong đoạn này, ta nghiên cứu một phương pháp tất định để ngắt tính

đối xứng. Điểm cốt lõi của thuật toán đó là sử dụng các chỉ số bộ xử lý hoặc các địa chỉ bộ nhớ thay vì các lần tung đồng xu ngẫu nhiên. Ví dụ, trong ví dụ hai-bộ xử lý, ta có thể ngắt tính đối xứng bằng cách cho phép bộ xử lý có chỉ số bộ xử lý nhỏ hơn đi trước—rõ ràng một tiến trình thời gian không đổi.

Ta sẽ dùng cùng ý tưởng, nhưng theo cách thông minh hơn nhiều, trong một thuật toán để ngắt tính đối xứng trong một danh sách nối kết n-đối tượng. Mục tiêu đó là chọn một phân số không đổi của các đối tượng trong danh sách nhưng tránh chọn hai đối tượng kề nhau. Có thể thực hiện thuật toán này với n bộ xử lý trong $O(\lg^* n)$ thời gian bằng một thuật toán tất định EREW. Bởi $\lg^* n \leq 5$ với tất cả $n \leq 2^{65536}$, giá trị $\lg^* n$ có thể được xem là một hằng nhỏ vì mọi mục tiêu thực tiễn (xem trang 36).

Thuật toán tất định của chúng ta gồm hai phần. Phần đầu tính toán một "tiến trình tô màu 6" của danh sách nối kết trong $O(\lg^* n)$ thời gian. Phần thứ hai chuyển đổi tiến trình tô màu 6 thành một "tập hợp độc lập tột độ" của danh sách trong $O(1)$ thời gian. Tập hợp độc lập tột độ sẽ chứa một phân số bất biến của n đối tượng của danh sách, và không có hai đối tượng trong tập hợp kề nhau.

Các tiến trình tô màu và các tập hợp độc lập tột độ

Một ***tiến trình tô màu*** của một đồ thị không hướng $G = (V, E)$ là một hàm $C : V \to \mathbf{N}$ sao cho với tất cả $u, v \notin V$, nếu $C(u) = C(v)$, thì $(u, v) \in E$; nghĩa là, không có các đỉnh kề nhau mang cùng màu. Trong một tiến trình tô màu 6 [6-coloring] của một danh sách nối kết, tất cả các màu nằm trong miền giá trị $\{0, 1, 2, 3, 4, 5\}$ và không có hai đỉnh liên tiếp có cùng màu. Thực vậy, mỗi danh sách nối kết có một tiến trình tô màu 2, bởi ta có thể tô màu các đối tượng có các hạng là lẻ bằng màu 0 và các đối tượng có các hạng chẵn bằng màu 1. Ta có thể tính toán một tiến trình tô màu như vậy trong $O(\lg n)$ thời gian dùng một phép tính tiền tố song song, nhưng với nhiều ứng dụng, ta chỉ cần tính toán một tiến trình tô màu $O(1)$. Ta sẽ chứng tỏ một tiến trình tô màu 6 có thể được tính toán trong $O(\lg^* n)$ thời gian mà không dùng ngẫu nhiên hóa.

Một ***tập hợp độc lập*** [independent set] của một đồ thị $G = (V, E)$ là một tập hợp con $V' \subseteq V$ của các đỉnh sao cho mỗi cạnh trong E liên thuộc trên tối đa một đỉnh trong V'. Một ***tập hợp độc lập tột độ***, hoặc ***MIS*** [maximal independent set], là một tập hợp độc lập V' sao cho với tất cả các đỉnh $v \in V - V'$, tập hợp $V' \cup \{v\}$ không độc lập—mọi đỉnh không nằm trong V' sẽ kề với một đỉnh nào đó trong V'. Đừng lẫn lộn bài toán tính toán một tập hợp độc lập *tột độ*—một bài toán dễ—với bài

toán tính toán một tập hợp độc lập cực đại [maximum independent set]—một bài toán khó. Bài toán tìm một tập hợp độc lập có kích cỡ cực đại trong một đồ thị chung là đầy đủ NP [NP-complete]. (Xem Chương 36 để biết chi tiết về tính đầy đủ NP. Bài toán 36-1 liên quan đến các tập hợp độc lập cực đại.)

Với các danh sách n-đối tượng, một tập hợp độc lập cực đại [maximum] (và vì vậy tột độ [maximal]) có thể được xác định trong $O(\lg n)$ thời gian bằng cách dùng một phép tính tiền tố song song, như trong tiến trình tô màu 2 vừa nêu, để định danh các đối tượng xếp hạng lẻ.

Phương pháp này lựa $\lceil n/2 \rceil$ đối tượng. Tuy nhiên, nhận thấy mọi tập hợp độc lập tột độ của một danh sách nối kết chứa ít nhất $n/3$ đối tượng, bởi với 3 đối tượng liên tiếp bất kỳ, ít nhất phải có một nằm trong tập hợp. Tuy nhiên, ta sẽ chứng tỏ một tập hợp độc lập cực đại của một danh sách có thể được xác định trong $O(1)$ thời gian căn cứ vào một tiến trình tô màu $O(1)$ của danh sách.

Tính toán một tiến trình tô màu 6

Thuật toán SIX-COLOR tính toán a tiến trình tô màu 6 của một danh sách. Ta sẽ gán mã giả của thuật toán, nhưng ta sẽ mô tả chi tiết về nó. Ta mặc nhận rằng thoạt đầu mỗi đối tượng x trong danh sách nối kết được kết hợp với một bộ xử lý riêng biệt $P(x) \in \{0, 1, ..., n-1\}$.

Ý tưởng của SIX-COLOR đó là tính toán một dãy $C_0[x]$, $C_1[x]$,..., $C_m[x]$ các màu cho mỗi đối tượng x trong danh sách. Tiến trình tô màu ban đầu C_0 là một tiến trình tô màu n. Mỗi lần lặp lại của thuật toán sẽ định nghĩa một tiến trình tô màu mới C_{k+1}, dựa trên tiến trình tô màu trước đó C_k, với $k = 0, 1, ..., m-1$. Tiến trình tô màu chung cuộc C_m là một tiến trình tô màu 6, và ta sẽ chứng minh $m = O(\lg^* n)$.

Tiến trình tô màu ban đầu là tiến trình tô màu n tầm thường ở đó $C_0[x] = P(x)$. Bởi không có hai đối tượng danh sách có cùng màu, nên không có hai đối tượng danh sách kề nhau có cùng màu, và do đó tiến trình tô màu là hợp pháp. Lưu ý, mỗi trong số các màu ban đầu có thể được mô tả bằng $\lceil \lg n \rceil$ bit, có nghĩa là nó có thể được lưu trữ trong một từ máy tính bình thường.

Các tiến trình tô màu sau đó có được như sau. Lần lặp lại thứ k, với $k = 0, 1, ..., m - 1$, bắt đầu bằng một tiến trình tô màu C_k và kết thúc bằng một tiến trình tô màu C_{k+1} dùng ít bit hơn cho mỗi đối tượng, như phần đầu của Hình 30.11 đã nêu. Giả sử rằng tại đầu của một lần lặp lại, màu C_k của mỗi đối tượng chiếm r bit. Ta xác định màu mới của một đối tượng x bằng cách tìm tới trong danh sách tại màu của $next[x]$.

Để chính xác hơn, giả sử rằng với mỗi đối tượng x, ta có $C_k[x] = a$ và $C_k[next[x]] = b$, ở đó $a = <a_{r-1}, a_{r-2},..., a_0>$ và $b = <b_{r-1}, b_{r-2},..., b_0>$ là các màu r bit. Bởi $C_k[x] \neq C_k[next[x]]$, ta có một chỉ số bé nhất i tại đó các bit của hai màu khác nhau: $a_i \neq b_i$. Bởi $0 \leq i \leq r-1$, nên ta có thể ghi i với chỉ $\lceil \lg r \rceil$ bit: $i = <i_{\lceil \lg r \rceil - 1}, i_{\lceil \lg r \rceil - 2},..., i_0>$. Ta tô màu lại x bằng giá trị của i được chắp với bit a_i. Nghĩa là, ta gán

$$C_{k+1}[x] = <i, a_i>$$
$$= <i_{\lceil \lg r \rceil - 1}, i_{\lceil \lg r \rceil - 2},..., i_0, a_i>.$$

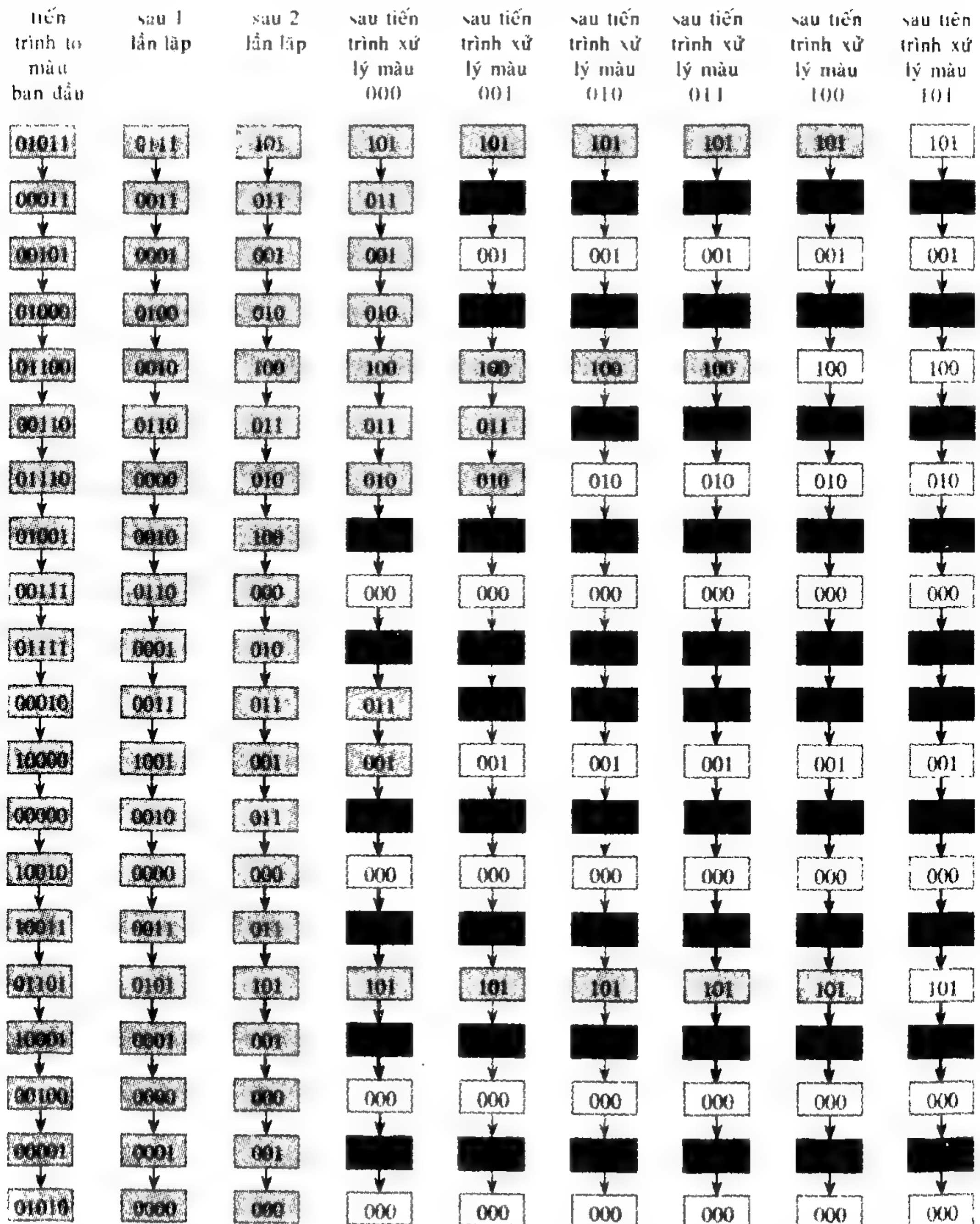
Đuôi của danh sách có màu mới $<0, a_0>$. Do đó, số lượng các bit trong mỗi màu mới tối đa là $\lceil \lg r \rceil + 1$.

Ta phải chứng tỏ nếu mỗi lần lặp lại của SIX-COLOR bắt đầu với một tiến trình tô màu, "tiến trình tô màu" mới mà nó tạo ra quả thật là một tiến trình tô màu hợp pháp. Để thực hiện điều này, ta chứng minh rằng $C_k[x] \neq C_k[next[x]]$ hàm ý $C_{k+1}[x] \neq C_{k+1}[next[x]]$. Giả sử rằng $C_k[x] = a$ và $C_k[next[x]] = b$, và rằng $C_{k+1}[x] = <i,a_i>$ và $C_{k+1}[next[x]] = <j, b_j>$. Có hai trường hợp để xét. Nếu $i \neq j$, thì $<i, a_i> \neq <j, b_j>$, và do đó các màu mới là khác nhau. Tuy nhiên, nếu $i = j$, thì $a_i \neq b_i = b_j$ theo phương pháp tô màu lại của chúng ta, và như vậy các màu mới một lần nữa lại khác nhau. (Tình huống tại đuôi danh sách cũng có thể được điều quản tương tự.)

Phương pháp tô màu lại mà SIX-COLOR sử dụng nhận một màu r bit và thay nó bằng một màu ($\lceil \lg r \rceil + 1$) bit, có nghĩa là số lượng bit hoàn toàn được rút gọn miễn là $r \geq 4$. Khi $r = 3$, hai màu có thể khác nhau trong vị trí bit 0, 1, hoặc 2. Do đó, mỗi màu mới là <00>, <01>, hoặc <10> được chắp với 0 hoặc 1, như vậy một lần nữa để lại một số 3 bit. Tuy nhiên, chỉ 6 trong số 8 giá trị khả dĩ của các con số 3 bit được dùng, sao cho SIX-COLOR quả thực kết thúc bằng một tiến trình tô màu 6.

Mặc nhận mỗi bộ xử lý có thể xác định chỉ số thích hợp i trong $O(1)$ thời gian và thực hiện một phép chuyển trái trong $O(1)$ thời gian—các phép toán thường được hỗ trợ trên nhiều máy thực tế—mỗi lần lặp lại mất $O(1)$ thời gian. Thủ tục SIX-COLOR là một thuật toán EREW: với mỗi đối tượng x, bộ xử lý của nó chỉ truy cập x và $next[x]$.

Cuối cùng, ta hãy xem tại sao chỉ cần $O(\lg^* n)$ lần lặp lại để mang tiến trình tô màu n ban đầu xuống thành một tiến trình tô màu 6. Ta đã định nghĩa $\lg^* n$ là số lần mà hàm lôga lg cần được áp dụng cho n để rút gọn nó tối đa là 1 hoặc, cho $\lg^{(i)} n$ thể hiện i ứng dụng liên tục của hàm lg.



Hình 30.11 Các thuật toán SIX-COLOR và LIST-MIS ngắt tính đối xứng trong một danh sách. Các thuật toán cùng nhau tìm ra một tập hợp lớn các đối tượng không liền kề trong $O(\lg^* n)$ thời gian dùng n bộ xử lý. Danh sách ban đầu của $n = 20$ đối tượng được nêu bên trái, chạy dọc. Mỗi đối tượng có một màu 5-bit ban đầu, riêng biệt. Với các tham số này, thuật toán SIX-COLOR chỉ cần hai lần lặp lại đã nêu để tô màu lại từng đối tượng bằng một màu trong miền {0, 1, 2, 3, 4 5}. Các đối tượng trắng được LIST-MIS đặt vào MIS khi các màu được xử lý, và các đối tượng đen bị triệt.

$$\lg^* n = \min\{i \ge 0 : \lg^{(i)} n \le 1\} .$$

Cho r_i là số lượng bit trong tiến trình tô màu ngay trước lần lặp lại thứ i. Bằng phương pháp quy nạp, ta chứng minh rằng nếu $\lceil \lg^{(i)} n \rceil \ge 2$, thì $r_i \le \lceil \lg^{(i)} n \rceil + 2$. Thoạt đầu, ta có $r_1 \le \lceil \lg n \rceil$. Lần lặp lại thứ i mang số lượng bit trong tiến trình tô màu xuống còn $r_{i+1} = \lceil \lg r_i \rceil + 1$. Giả sử giả thuyết quy nạp áp dụng cho r_{i-1}, ta được

$$\begin{aligned}
r_i &= \lceil \lg r_{i-1} \rceil + 1 \\
&\le \lceil \lg(\lceil \lg^{(i-1)} n \rceil + 2) \rceil + 1 \\
&\le \lceil \lg(\lg^{(i-1)} n + 3) \rceil + 1 \\
&\le \lceil \lg(2 \lg^{(i-1)} n) \rceil + 1 \\
&= \lceil \lg(\lg^{(i-1)} n) + 1 \rceil + 1 \\
&= \lceil \lg^{(i)} n \rceil + 2 .
\end{aligned}$$

Dòng thứ tư là do giả thiết cho rằng $\lceil \lg^{(i)} n \rceil \ge 2$, có nghĩa là $\lceil \lg^{(i-1)} n \rceil \ge 3$. Do đó, sau $m = \lg^* n$ bước, số lượng bit trong tiến trình tô màu là $r_m \le \lceil \lg^{(m)} n \rceil + 2 = 3$, bởi $\lg^{(m)} n \le 1$ theo định nghĩa của hàm $\lg^*$. Như vậy, tối đa thêm một lần lặp lại cũng đủ để tạo ra một tiến trình tô màu 6. Do đó, tổng thời gian của SIX-COLOR là $O(\lg^* n)$.

Tính toán một MIS từ một tiến trình tô màu 6

Tiến trình tô màu là phần khó của tiến trình ngắt tính đối xứng. Thuật toán EREW LIST-MIS sử dụng n bộ xử lý để tìm ra một tập hợp độc lập tột độ trong $O(c)$ thời gian căn cứ vào một tiến trình tô màu c của một danh sách n đối tượng. Như vậy, sau khi tính toán một tiến trình tô màu 6 của một danh sách, ta có thể tìm một tập hợp độc lập tột độ của danh sách nối kết trong $O(1)$ thời gian.

Phần sau của Hình 30.11 minh họa ý tưởng đàng sau LIST-MIS. Ta có một tiến trình tô màu c C. Với mỗi đối tượng x, ta lưu giữ một bit *alive*$[x]$, báo cho biết x vẫn là một ứng viên để gộp trong MIS. Thoạt đầu, *alive*$[x]$ = TRUE với tất cả đối tượng x.

Sau đó, thuật toán lặp lại qua mỗi trong số c màu. Trong lần lặp lại của màu i, mỗi bộ xử lý chịu trách nhiệm một đối tượng x sẽ kiểm tra $C[x] = i$ và *alive*$[x]$ = TRUE hay không. Nếu cả hai điều kiện đứng vững, thì bộ xử lý đánh dấu x là thuộc về MIS đang được kiến tạo. Tất cả các đối tượng kề với các đối tượng được bổ sung vào MIS—những đối tượng đứng ngay trước hoặc theo sau—đều có các bit *alive* của chúng được ấn định là FALSE; chúng không thể nằm trong MIS bởi chúng kề với một đối tượng trong MIS. Sau tất cả c lần lặp lại, mỗi đối tượng hoặc đạ bị "triệt"—bit *alive* của nó đã được ấn định là FALSE—hoặc được đặt vào MIS.

Ta phải chứng tỏ tập hợp kết quả là độc lập và tột độ. Để xem nó là

độc lập, ta giả sử rằng hai đối tượng kề x và $next[x]$ được đặt vào tập hợp. Bởi chúng kề nhau, nên $C[x] \neq C[next[x]]$, bởi C là một tiến trình tô màu. Không để mất tính tổng quát, ta mặc nhận rằng $C[x] < C[next[x]]$, sao cho x được đặt vào tập hợp trước khi đặt $next[x]$. Nhưng khi đó $alive[next[x]]$ đã được ấn định là FALSE vào lúc các đối tượng của màu $C[next[x]]$ được xét, và $next[x]$ đã không thể được đưa vào tập hợp.

Để xem tập hợp là tột độ, ta giả sử rằng không có đối tượng nào trong số ba đối tượng liên tiếp x, y, và z được đưa vào tập hợp. Tuy vậy, cách duy nhất để y có thể tránh bị đưa vào tập hợp, đó là nếu nó đã bị triệt khi một đối tượng kề cận đã được đưa vào tập hợp. Bởi, theo giả thiết của chúng ta, cả x lẫn z đều không được đưa vào tập hợp, nên đối tượng y vẫn phải sống vào lúc các đối tượng của màu $C[y]$ được xử lý. Nó phải được đưa vào MIS.

Mỗi lần lặp lại của LIST-MIS mất $O(1)$ thời gian trên một PRAM. Thuật toán là EREW bởi mỗi đối tượng chỉ truy cập chính nó, phần tử tiền vị của nó, và phần tử kế vị của nó trong danh sách. Tổ hợp LIST-MIS với SIX-COLOR, ta có thể ngắt tính đối xứng trong một danh sách nối kết trong $O(\lg^* n)$ thời gian một cách tất định.

Bài tập

30.5-1

Với ví dụ về ngắt tính đối xứng 2 bộ xử lý tại đầu đoạn này, hãy chứng tỏ tính đối xứng được ngắt trong thời gian dự trù bất biến.

30.5-2

Cho một tiến trình tô màu 6 của một danh sách n-đối tượng, nêu cách tô màu 3 danh sách trong $O(1)$ thời gian dùng n bộ xử lý trong một PRAM EREW.

30.5-3

Giả sử mọi nút phi gốc trong một cây n-nút có một biến trỏ đến cha củanó. Nêu một thuật toán CREW để tô $O(1)$-màu cho cây trong $O(\lg^* n)$ thời gian.

30.5-4 *

Nêu một thuật toán PRAM hiệu quả để tô $O(1)$-màu cho một đồ thị độ-3. Phân tích thuật toán của bạn.

30.5-5

Một ***tập hợp đường sinh k*** [k-ruling set] của một danh sách nối kết là một tập hợp các đối tượng (các thước) trong danh sách sao cho không có thước nào kề nhau và tối đa k phi thước (các chủ thể [subjects]) tách biệt

các thước. Như vậy, một MIS là một tập hợp 2 đường sinh. Nêu cách tính một tập hợp $O(\lg n)$ đường sinh của một danh sách n đối tượng trong $O(1)$ thời gian dùng n bộ xử lý. Nêu cách tính một tập hợp $O(\lg \lg n)$ đường sinh trong $O(1)$ thời gian theo các giả thiết tương tự.

30.5-6 *

Nêu cách tìm một tiến trình tô màu 6 của một danh sách nối kết n-đối tượng trong $O(\lg(\lg^* n))$ thời gian. Giả sử mỗi bộ xử lý có thể lưu trữ một bảng tính trước có kích cỡ $O(\lg n)$. (*Mách nước*: Trong SIX-COLOR, màu chung cuộc của một đối tượng tùy thuộc vào bao nhiêu giá trị?)

Các Bài Toán

30-1 Tiền tố song song phân đoạn

Cũng như một phép tính tiền tố bình thường, một ***phép tính tiền tố phân đoạn*** được định nghĩa theo dạng một toán tử kết hợp, nhị phân, ⊗. Nó nhận một dãy đầu vào $x = \langle x_1, x_2, ..., x_n\rangle$ có các thành phần được rút từ một miền xác định S và một dãy ***phân đoạn*** $b = \langle b_1, b_2, ..., b_n\rangle$ có các thành phần được rút từ miền xác định $\{0,1\}$, với $b_1 = 1$. Nó tạo ra một dãy kết xuất $y = \langle y_1, y_2, ..., y_n\rangle$ trên miền xác định S. Các bit của b xác định một tiến trình phân hoạch x và y thành các phân đoạn; một phân đoạn mới bắt đầu mọi nơi $b_i = 1$, và phân đoạn hiện hành tiếp tục nếu $b_i = 0$. Phép tính tiền tố phân đoạn thực hiện một phép tính tiền tố độc lập trong mỗi phân đoạn của x để tạo ra phân đoạn tương ứng của y. Hình 30.12 minh họa một phép tính tiền tố phân đoạn dùng phép cộng bình thường.

a. Định nghĩa toán tử $\hat{\otimes}$ trên các cặp có thứ tự $(a, z), (a', z') \in \{0, 1\} \times S$ như sau:

$$(a, z)\ \hat{\otimes}\ (a', z') = \begin{cases} (a, z \otimes z') & \text{nếu } a' = 0\,, \\ (1, z') & \text{nếu } a' = 1\,. \end{cases}$$

Chứng minh $\hat{\otimes}$ là kết hợp.

b. Nêu cách thực thi các phép tính tiền tố phân đoạn trên một danh sách n thành phần trong $O(\lg n)$ thời gian trên một PRAM EREW.

c. Mô tả một thuật toán EREW $O(k \lg n)$ thời gian để sắp xếp một danh sách n số k-bit.

30-2 Thuật toán cực đại hiệu quả bộ xử lý

Ta muốn tìm cực đại của n số trên một PRAM CRCW với $p = n$ bộ xử lý.

a. Chứng tỏ bài toán tìm cực đại của $m \le p/2$ số có thể được rút gọn thành bài toán tìm cực đại của tối đa m^2/p số trong $O(1)$ thời gian trên một PRAM CRCW p-bộ xử lý.

b. Nếu ta bắt đầu với $m = \lfloor p/2 \rfloor$ số, có bao nhiêu số còn lại sau k lần lặp lại của thuật toán trong phần (a)?

b =	1	0	0	1	0	1	1	0	0	0	0	0	1	0
x =	1	2	3	4	5	6	7	8	9	10	11	12	13	14
y =	1	3	6	4	9	6	7	15	24	34	45	57	13	27

Hình 30.12 Một phép tính tiền tố phân đoạn với dãy phân đoạn b, dãy đầu vào x, và dãy kết xuất y. Có 5 phân đoạn.

c. Chứng tỏ bài toán tìm cực đại của n số có thể được giải trong $O(\lg \lg n)$ thời gian trên một PRAM CROW có $p = n$ bộ xử lý.

30-3 Các thành phần liên thông

Trong bài toán này, ta nghiên cứu một thuật toán CRCW tùy ý để tính toán các thành phần liên thông của một đồ thị không hướng $G = (V, E)$ sử dụng $|V + E|$ bộ xử lý. Cấu trúc dữ liệu được dùng là một rừng tập hợp rời (xem Đoạn 22.3). Mỗi đỉnh $v \in V$ duy trì một biến trỏ $p(v)$ đến một cha. Thoạt đầu, $p[v] = v$: đỉnh trỏ đến chính nó. Vào cuối thuật toán, với hai đỉnh bất kỳ $u, v \in V$, ta có $p[u] = p[v]$ nếu và chỉ nếu $u \leadsto v$ trong G. Trong khi chạy thuật toán, p biến trỏ hình thành một rừng các cây ***biến trỏ*** có gốc. Một ***sao*** [star] là một cây biến trỏ ở đó $p[u] = p[v[$ với tất cả các đỉnh u và v trong cây.

Thuật toán các thành phần liên thông mặc nhận mỗi cạnh $(u, v) \in E$ xuất hiện hai lần: một lần dưới dạng (u, v) và một lần dưới dạng (v, u). Thuật toán sử dụng hai phép toán cơ bản, HOOK và JUMP, và một chương trình con STAR ấn định $star[v]$ = TRUE nếu v thuộc về một sao.

```
Hoox(G)
STAR(G)
for mỗi cạnh (u, v) ∈ E[G], song song
        do if  star[u] và p[u] > p[v]
                 then p[p[u]) ← p[v]
STAR(G)
for mỗi cạnh (u, v) ∈ E[G], song song
        do if  star[u) và p[u] ≠ p[v]
                 thì p[p[u]]  ←  p[v]
JUMP(G)
for mỗi v ∈ V[G], song song
        do p[v] ←  p[p[v]]
```

Thuật toán các thành phần liên thông thực hiện một HOOK ban đầu,

rồi nó liên tục thực hiện HOOK, JUMP, HOOK, JUMP, và vân vân, cho đến khi không có biến trỏ nào bị một phép toán JUMP thay đổi. (Lưu ý, hai HOOK được thực hiện trước JUMP đầu tiên.)

a. Nêu mã giả của STAR(G).

b. Chứng tỏ p biến trỏ quả thực hình thành các cây có gốc, với gốc của một cây trỏ đến chính nó. Chứng tỏ nếu u và v nằm trong cùng cây biến trỏ, thì $u \leadsto v$ trong G.

c. Chứng tỏ thuật toán là đúng: nó kết thúc, và khi nó kết thúc, $p[u] = p[v]$ nếu và chỉ nếu $u \leadsto v$ trong G.

Để phân tích thuật toán các thành phần liên thông, ta xét một thành phần liên thông đơn giản C, mà mặc nhận có ít nhất hai đỉnh. Giả sử tại một điểm nào đó trong khi chạy thuật toán, C được cấu tạo bởi một tập hợp $\{T_i\}$ cây biến trỏ. Định nghĩa thế của C là

$$\Phi(C) = \sum_{T_i} \text{height}(T_i) .$$

Mục tiêu của cuộc phân tích đó là chứng minh mỗi lần lặp lại của tiến trình móc ráp [hooking] và nhảy [jumping] sẽ làm $\Phi(C)$ giảm theo một thừa số bất biến.

d. Chứng minh sau HOOK ban đầu, không có cây biến trỏ nào có chiều cao 0 và $\Phi(C) \le |V|$.

e. Chứng tỏ sau HOOK ban đầu, các phép toán HOOK tiếp theo không bao giờ gia tăng $\Phi(C)$.

f. Chứng tỏ sau mọi phép toán HOOK phi ban đầu, không có cây biến trỏ nào là một sao trừ phi cây biến trỏ chứa tất cả các đỉnh trong C.

g. Chứng tỏ nếu C không được co cụm thành một sao đơn lẻ, thì sau một phép toán JUMP, $\Phi(C)$ tối đa bằng 2/3 giá trị trước đó của nó. Minh họa ca xấu nhất.

h. Kết luận thuật toán xác định tất cả các thành phần liên thông của G trong $O(\lg V)$ thời gian.

30-4 Chuyển vị một ảnh mành

Một vùng đệm khung đồ họa mành [raster-graphics] có thể được xem là một ma trận $p \times p$ M gồm các bit. Phần cứng hiển thị đồ họa mành khiến ma trận con trên trái $n \times n$ của M lộ diện trên màn hình người dùng. Một phép toán BITBLT (BLock Transfer of BITs) được dùng để dời một hình chữ nhật gồm các bit từ vị trí này sang vị trí khác. Cụ thể, BITBLT(r_1, c_1, r_2, c_2, nr, nc, *) ấn định

$$M[r_2 + i, c_2 + j] \leftarrow M[r_2 + i, c_2 + j] * M[r_1 + i, c_1 + j]$$

với $i = 0, 1, ..., nr - 1$ và $j = 0, 1, ..., nc - 1$, ở đó * là bất kỳ trong số

16 hàm bool trên hai đầu vào.

Ta quan tâm đến việc chuyển vị ảnh ($M[i, j] \leftarrow M[j, i]$) trong phần lộ diện của vùng đệm khung. Ta mặc nhận mức hao phí của tiến trình chép các bit nhỏ hơn mức hao phí của tiến trình gọi nguyên hàm BITBLT, và do đó ta quan tâm đến việc dùng càng ít phép toán BITBLT càng tốt.

Chứng tỏ bất kỳ ảnh nào trên màn hình đều có thể được chuyển vị với $O(\lg n)$ phép toán BITBLT. Giả sử p đủ lớn hơn n sao cho phần không lộ diện của vùng đệm khung cung cấp đủ kho lưu trữ làm việc. Bạn cần thêm bao nhiêu kho lưu trữ bổ sung? (*Mách nước*: Dùng cách tiếp cận chia để trị song song ở đó vài trong số các BITBLT được thực hiện bằng các AND bool.)

Ghi chú Chương

Akl [9], Karp và Ramachandran [118), và Leighton [135] nghiên cứu các thuật toán song song cho các bài toán tổ hợp. Các kiến trúc máy song song khác nhau được mô tả bởi Hwang và Briggs [109], Hwang và DeGroot [110].

Lý thuyết tính toán song song bắt đầu vào cuối những năm 1940 khi J. Von Neumann [38] giới thiệu một mô hình hạn chế của tính toán song song có tên otomat tế bào, mà chủ yếu là một mảng hai chiều các bộ xử lý trạng thái hữu hạn được tương kết theo dạng hình lưới. Mô hình PRAM đã được Fortune và Wyllie [73] chính thức hóa vào năm 1978, mặc dù trước đó đã có nhiều tác giả khác mô tả các mô hình mà về cơ bản cũng tương tự.

Kỹ thuật nhảy biến trỏ đã được Wyllie [204] giới thiệu. Cuộc nghiên cứu các phép tính tiền tố song song đã nảy sinh từ công trình của Ofman [152] trong ngữ cảnh của phép cộng nhớ kiểm tra trước. Kỹ thuật tua Euler là do Tarjan và Vishkin [191).

Các mức trả giá về thời gian bộ xử lý để tính toán cực đại của một tập hợp n con số đã được Valiant [193] cung cấp, ông cũng đã chứng tỏ một thuật toán hiệu quả công $O(1)$ thời gian không tồn tại. Cook, Dwork, và Reischuk [50] đã chứng minh bài toán tính toán cực đại đòi hỏi $\Omega(\lg n)$ thời gian trên một PRAM CREW. Phép mô phỏng một thuật toán CRCW bằng thuật toán EREW là của Vishkin [195].

Định lý 30.2 là do Brent [34]. Thuật toán ngẫu nhiên hóa để xếp hạng danh sách hiệu quả công đã được Anderson và Miller [11] khám phá. Họ cũng có một thuật toán hiệu quả công, tất định, cho cùng bài toán [10]. Thuật toán về ngắt tính đối xứng tất định là của Goldberg và Plotkin [84]. Nó dựa trên một thuật toán tương tự với cùng thời gian thực hiện của Cole và Vishkin [47].

31 Các phép toán ma trận

Các phép toán trên các ma trận là trung tâm của việc tính toán khoa học. Do đó, các thuật toán hiệu quả để làm việc với các ma trận được quan tâm đáng kể về mặt thực tiễn. Chương này giới thiệu ngắn gọn về lý thuyết ma trận và các phép toán ma trận, nhấn mạnh các bài toán nhân các ma trận và giải các tập hợp của các phương trình tuyến tính đồng thời.

Sau khi Đoạn 31.1 giới thiệu các khái niệm ma trận căn bản và các hệ ký hiệu, Đoạn 31.2 trình bày thuật toán đáng ngạc nhiên của Strassen để nhân hai ma trận n x n trong $\Theta(n^{\lg 7}) = O(n^{2.81})$ thời gian. Đoạn 31.3 định nghĩa các tựa vành, các vành, và các trường, làm rõ giả thiết cần có để khiến thuật toán Strassen làm việc. Nó cũng chứa một thuật toán nhanh theo tiệm cận để nhân các ma trận bool. Đoạn 31.4 nêu cách giải một tập hợp các phương trình tuyến tính dùng các phép phân tích LUP. Sau đó, Đoạn 31.5 khảo sát mối quan hệ mật thiết giữa bài toán nhân các ma trận và bài toán đảo một ma trận. Cuối cùng, Đoạn 31.6 mô tả lớp quan trọng của các ma trận xác định dương đối xứng và nêu cách dùng chúng để tìm một giải pháp các bình phương bé nhất cho một tập hợp xác định trên [overdetermined set] gồm các phương trình tuyến tính.

31.1 Các tính chất của các ma trận

Trong đoạn này, ta ôn lại vài khái niệm căn bản về lý thuyết ma trận và vài tính chất căn bản của các ma trận, tập trung vào những tính chất cần thiết trong các đoạn về sau.

Các ma trận và các vectơ

Một ***ma trận*** [matrix] là một mảng chữ nhật gồm các con số. Ví dụ,

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \quad (31.1)$$

là một ma trận 2 x 3 $A = (a_{ij})$, ở đó với $i = 1, 2$ và $j = 1, 2, 3$, thành phần của ma trận trong hàng i và cột j là a_{ij}. Ta dùng các mẫu tự hoa để thể hiện các ma trận và các mẫu tự thường tương ứng viết dưới để thể hiện các thành phần của chúng. Tập hợp tất cả các ma trận $m \times n$ có các khoản nhập giá trị thực sẽ được ký hiệu là $\mathbf{R}^{m \times n}$. Nói chung, tập hợp các ma trận $m \times n$ có các khoản nhập rút từ một tập hợp S sẽ được ký hiệu là $S^{m \times n}$.

Chuyển vị của một ma trận A là ma trận A^T có được bằng cách trao đổi các hàng và các cột của A. Với ma trận A của phương trình (31.1),

$$A^T = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}$$

Một ***vectơ*** là một mảng một chiều các con số. Ví dụ,

$$x = \begin{pmatrix} 2 \\ 3 \\ 5 \end{pmatrix} \qquad (31.2)$$

là một vectơ có kích cỡ 3. Ta dùng các mẫu tự viết thường để thể hiện các vectơ, và để thể hiện thành phần thứ i của một vectơ x có kích cỡ n, ta dùng ký hiệu x_i, với $i = 1, 2, ..., n$. Ta quy cho dạng chuẩn của một vectơ là một ***vectơ cột*** tương đương với một ma trận $n \times 1$; ***vectơ hàng*** tương ứng có được bằng cách lấy chuyển vị:

$x^T = (2 \quad 3 \quad 5)$.

Vectơ đơn vị e_i là vectơ có thành phần thứ i là 1 và tất cả các thành phần khác của nó là 0. Nói chung, kích cỡ của một vectơ đơn vị thường đã rõ theo ngữ cảnh.

Ma trận zero là một ma trận có mọi khoản nhập là 0. Một ma trận như vậy thường được thể hiện bằng 0, bởi sự mơ hồ giữa con số 0 và một ma trận các 0 thường dễ dàng giải theo ngữ cảnh. Nếu một ma trận các 0 là có chủ đích, thì kích cỡ của ma trận cũng cần được phái sinh theo ngữ cảnh.

*Các **ma trận bình phương*** $n \times n$ thường nảy sinh. Có vài trường hợp đặc biệt về các ma trận bình phương cần được quan tâm cụ thể:

1. Một ***ma trận đường chéo*** có $a_{ij} = 0$ mỗi khi $i \neq j$. Bởi vì tất cả các thành phần ngoài đường chéo [off-diagonal] là zero, có thể chỉ định ma trận bằng cách liệt kê các thành phần dọc theo đường chéo:

$$\text{diag}(a_{11}, a_{22}, \ldots, a_{nn}) = \begin{pmatrix} a_{11} & 0 & \ldots & 0 \\ 0 & a_{22} & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & a_{nn} \end{pmatrix}$$

2. ***Ma trận đồng nhất thức*** $n \times n$ I_n là một ma trận đường chéo có các 1 nằm dọc theo đường chéo:

$$I_n = \text{diag}(1, 1, \ldots, 1)$$

$$= \begin{pmatrix} 1 & 0 & \ldots & 0 \\ 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 0 \end{pmatrix}.$$

Khi I xuất hiện mà không có con chữ dưới, kích cỡ củanó có thể được phái sinh theo ngữ cảnh. Cột thứ i của một ma trận đồng nhất thức là vectơ đơn vị e_i.

3. Một ***ma trận ba đường chéo*** [tridiagonal matrix] T là một ma trận mà $t_{ij} = 0$ nếu $|i - j| > 1$. Các khoản nhập phi zero chỉ xuất hiện trên đường chéo chính, ngay bên trên đường chéo chính ($t_{i,i+1}$ for $i = 1, 2, \ldots, n-1$), hoặc ngay bên dưới đường chéo chính ($t_{i+1,i}$ với $i = 1, 2, \ldots, n-1$):

$$T = \begin{pmatrix} t_{11} & t_{12} & 0 & 0 & \ldots & 0 & 0 & 0 \\ t_{21} & t_{22} & t_{23} & 0 & \ldots & 0 & 0 & 0 \\ 0 & t_{32} & t_{33} & t_{34} & \ldots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \ldots & t_{n-2,n-2} & t_{n-2,n-1} & 0 \\ 0 & 0 & 0 & 0 & \ldots & t_{n-1,n-2} & t_{n-1,n-1} & t_{n-1,n} \\ 0 & 0 & 0 & 0 & \ldots & 0 & t_{n,n-1} & t_{nn} \end{pmatrix}.$$

4. Một ***ma trận tam giác trên*** U là một ma trận mà $u_{ij} = 0$ nếu $i > j$. Tất cả các khoản nhập bên dưới đường chéo là zero:

$$U = \begin{pmatrix} u_{11} & u_{12} & \ldots & u_{1n} \\ 0 & u_{22} & \ldots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & u_{nn} \end{pmatrix}$$

Một ma trận tam giác trên là ***tam giác trên đơn vị*** nếu nó có tất cả các 1 nằm dọc theo đường chéo.

5. Một ***ma trận tam giác dưới*** L là một ma trận mà $l_{ij} = 0$ nếu $i < j$.

Tất cả các khoản nhập bên trên đường chéo là zero:

$$L = \begin{pmatrix} l_{11} & 0 & \dots & 0 \\ l_{21} & l_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \dots & l_{nn} \end{pmatrix}$$

Một ma trận tam giác dưới là ***tam giác dưới đơn vị*** nếu nó có tất cả các 1 nằm dọc theo đường chéo.

6. Một ***ma trận hoán vị*** P có chính xác một 1 trong mỗi hàng hoặc cột, và các 0 ở chỗ khác. Một ví dụ về ma trận hoán vị là

$$P = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Sở dĩ một ma trận như vậy được gọi là một ma trận hoán vị bởi vì việc nhân một vectơ x với một ma trận hoán vị có hiệu ứng hoán vị (dàn xếp lại) các thành phần của x.

7. Một ***ma trận đối xứng*** A thỏa điều kiện $A = A^{T}$. Ví dụ,

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 6 & 4 \\ 3 & 4 & 5 \end{pmatrix}$$

là một ma trận đối xứng.

Các phép toán trên ma trận

Các thành phần của một ma trận hoặc vectơ là các con số từ một hệ thống số, như các số thực, các số phức, hoặc các số nguyên modulo một số nguyên tố. Hệ thống số định nghĩa cách cộng và nhân các con số. Ta có thể mở rộng các phần định nghĩa này để bao hàm phép cộng và phép nhân của các ma trận.

Ta định nghĩa ***phép cộng ma trận*** như sau. Nếu $A = (a_{ij})$ và $B = (b_{ij})$ là các ma trận $m \times n$, thì tổng ma trận $C = (c_{ij}) = A + B$ của chúng là ma trận $m \times n$ được định nghĩa bởi

$$c_{ij} = a_{ij} + b_{ij}$$

với $i = 1, 2, \dots, m$ và $j = 1, 2, \dots, n$. Nghĩa là, phép cộng ma trận được thực hiện ở cấp thành phần. Một ma trận zero là đồng nhất thức cho phép cộng ma trận:

$$A + 0 = A$$
$$= 0 + A\ .$$

Nếu λ là một số và $A = (a_{ij})$ là một ma trận, thì $\lambda A = (\lambda a_{ij})$ là ***bội vô hướng*** của λ có được bằng cách nhân mỗi trong số các thành phần của nó với A. Là một trường hợp đặc biệt, ta định nghĩa ***âm*** của một ma trận $A = (a_{ij})$ là $-1 \cdot A = -A$, sao cho khoản nhập thứ *ij* của $-A$ là $-a_{ij}$. Như vậy,

$$A + (-A) = A$$
$$= 0 + A\ .$$

Căn cứ vào định nghĩa này, ta có thể định nghĩa phép trừ ma trận là phép cộng bản âm của một ma trận: $A - B = A + (-B)$.

Ta định nghĩa ***phép nhân ma trận*** như sau. Ta bắt đầu với hai ma trận *A* và *B* ***tương thích*** theo nghĩa là số lượng các cột của *A* bằng số lượng các hàng của *B*. (Nói chung, một biểu thức chứa một tích ma trận *AB* luôn được mặc nhận để hàm ý rằng các ma trận *A* và *B* là tương thích.) Nếu $A = (a_{ij})$ là một ma trận $m \times n$ và $B = (b_{jk})$ là một ma trận $n \times p$, thì tích ma trận của chúng $C = AB$ là ma trận $m \times p$ $C = (c_{ik})$, ở đó

$$c_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk} \qquad (31.3)$$

với $i = 1, 2, \ldots, m$ và $k = 1, 2, \ldots, p$. Thủ tục MATRIX-MULTIPLY trong Đoạn 26.1 thực thi phép nhân ma trận theo cách đơn giản dựa trên phương trình (31.3), giả định các ma trận là bình phương: $m = n = p$. Để nhân các ma trận $n \times n$, MATRIX-MULTIPLY thực hiện n^3 phép nhân và $n^2(n-1)$ phép cộng, và thời gian thực hiện của nó là $\Theta(n^3)$.

Các ma trận có nhiều (nhưng không phải tất cả) tính chất đại số điển hình của các con số. Các ma trận đồng nhất thức là các đồng nhất thức cho phép nhân ma trận:

$$I_m A = A I_n = A$$

với bất kỳ ma trận $m \times n$ *A*. Nhân với một ma trận zero sẽ cho ra một ma trận zero:

$$A0 = 0.$$

Phép nhân ma trận là kết hợp:

$$A(BC) = (AB)C \qquad (31.4)$$

với các ma trận tương thích *A*, *B*, và *C*. Phép nhân ma trận phân phối trên phép cộng:

$$A(B + C) = AB + AC\ .$$

$$(B + C)D = BD + CD . \quad (31.5)$$

Tuy nhiên, phép nhân của các ma trận $n \times n$ không giao hoán, trừ phi $n = 1$.

Ví dụ, nếu $A = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$ và $B = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$, thì

$$AB = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$$

và

$$BA = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} .$$

Các tích ma trận-vectơ hoặc các tích vectơ-vectơ được định nghĩa như thể vectơ là ma trận $n \times 1$ tương đương (hoặc một ma trận $1 \times n$, trong trường hợp của một vectơ hàng). Như vậy, nếu A là một ma trận $m \times n$ và x là một vectơ có kích cỡ n, thì Ax là một vectơ có kích cỡ m. Nếu x và y là các vectơ có kích cỡ n, thì n

$$x^T y = \sum_{i=1}^{n} x_i y_i$$

là một số (thực tế là một ma trận 1×1) có tên là ***tích trong*** của x và y. Ma trận xy^T là một ma trận $n \times n$ Z có tên là ***tích ngoài*** của x và y, với $z_{ij} = x_i y_j$. ***Quy mẫu*** [norm] (***euclid***) $\|x\|$ của một vectơ x có kích cỡ n được định nghĩa bởi

$$\begin{aligned} \|x\| &= (x_1^2 + x_2^2 + \dots + x_n^2)^{1/2} \\ &= (x^T x)^{1/2} . \end{aligned}$$

Như vậy, quy mẫu của x là chiều dài của nó trong không gian euclid n chiều.

Các nghịch đảo ma trận, các hạng, và các định thức

Ta định nghĩa ***nghịch đảo*** của một ma trận $n \times n$ A là ma trận $n \times n$, được thể hiện bằng A^{-1} (nếu nó tồn tại), sao cho $AA^{-1} = I_n = A^{-1}A$. Ví dụ,

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{-1} = \begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix} .$$

Nhiều ma trận phi zero $n \times n$ không có các nghịch đảo. Một ma trận mà không có nghịch đảo được gọi là ***không nghịch đảo*** [noninversible], hoặc ***lập dị*** [singular]. Một ví dụ về một ma trận lập dị phi zero là

$$\begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}.$$

Nếu một ma trận có một nghịch đảo, nó được gọi là ***invertible***, hoặc ***phi lập dị*** [nonsingular]. Khi tồn tại, các nghịch đảo ma trận là duy nhất. (Xem Bài tập 31.1-4.) Nếu A và B là các ma trận phi lập dị $n \times n$, thì

$$(BA)^{-1} = A^{-1}B^{-1}, \qquad (31.6)$$

Phép toán nghịch đảo giao hoán với phép toán chuyển vị:

$$(A^{-1})^{T} = (A^{T})^{-1}.$$

Các vectơ $x_1, x_2, ..., x_n$ là ***lệ thuộc tuyến tính*** [linearly dependent] nếu ở đó tồn tại các hệ số $c_1, c_2, ..., c_n$, không phải tất cả những hệ số zero, sao cho $c_1x_1 + c_2x_2 + ... + c_nx_n = 0$. Ví dụ, các vectơ $x_1 = (1\ 2\ 3)^T$, $x_2 = (2\ 6\ 4)^T$ và $x_3 = (4\ 11\ 9)^T$ là lệ thuộc tuyến tính, bởi $2x_1 + 3x_2 - 2x_3 = 0$. Nếu các vectơ không lệ thuộc tuyến tính, chúng là ***độc lập tuyến tính***. Ví dụ, các cột của một ma trận đồng nhất thức là độc lập tuyến tính.

Hạng cột [column rank] của một ma trận phi zero $m \times n$ A là kích cỡ của tập hợp lớn nhất gồm các cột bậc lập tuyến tính của A. Cũng vậy, ***hạng hàng*** của A là kích cỡ của tập hợp lớn nhất của các hàng bậcc lập tuyến tính của A. Một tính chất căn bản của một ma trận A bất kỳ đó là hạng hàng của nó luôn bằng hạng cột của nó, sao cho ta có thể đơn giản tham chiếu ***hạng*** của A. Hạng của một ma trận $m \times n$ là một số nguyên giữa 0 và min(m, n), kể cả các giá trị này. (Hạng của một ma trận zero là 0, và hạng của một ma trận đồng nhất thức $n \times n$ là n.) Một định nghĩa thay thế , nhưng tương đương và thường hữu ích hơn đó là hạng của một ma trận phi zero $m \times n$ A là số nhỏ nhất r sao cho ở đó tồn tại các ma trận B và C của các kích cỡ tương ứng $m \times r$ và $r \times n$ sao cho

$$A = BC.$$

Một ma trận bình phương $n \times n$ có ***hạng đầy đủ*** [full rank] nếu hạng của nó là n. Một tính chất căn bản của các hạng được căn cứ vào định lý dưới đây.

Định lý 31.1

Một ma trận bình phương có hạng đầy đủ nếu và chỉ nếu nó là phi lập dị.

Một ma trận $m \times n$ có ***hạng cột đầy đủ*** nếu hạng của nó là n.

Một ***vectơ rỗng*** cho một ma trận A là một vectơ phi zero x sao cho $Ax = 0$. Định lý dưới đây, mà phần chứng minh được chừa lại làm Bài tập 31.1-8, và hệ luận của nó liên kết các khái niệm của hạng cột và điểm lập dị với các vectơ rỗng.

***Định lý* 31.2**

Một ma trận A có hạng cột đầy đủ nếu và chỉ nếu nó không có một vectơ rỗng.

***Hệ luận* 31.3**

Một ma trận bình phương A là lập dị nếu và chỉ nếu nó có một vectơ rỗng.

Định thức con [minor] thứ ij của một ma trận $n \times n$ A, với $n > 1$, là ma trận $(n - 1) \times (n - 1)$ $A_{[ij]}$ có được bằng cách xóa hàng thứ i và cột thứ j của A. ***Định thức*** [determinant] của một ma trận $n \times n$ A có thể được định nghĩa một cách đệ quy theo dạng các định thức con của nó bởi

$$\det(A) = \begin{cases} a_{11} & \text{nếu } n = 1 , \\ a_{11} \det(A_{[11]}) - a_{12} \det(A_{[12]}) & \\ + \dots + (-1)^{n+1} a_{1n} \det(A_{[1n]}) & \text{nếu } n > 1 . \end{cases} \quad (31.7)$$

Số hạng $(-1)^{i+j} \det(A_{[ij]})$ được xem là ***phần phụ đại số*** [cofactor] của thành phần a_{ij}.

Các định lý dưới đây, mà chứng minh được bỏ qua ở đây, sẽ diễn tả các tính chất căn bản của định thức.

Định lý 31.4 (Các tính chất định thức)

Định thức của một ma trận bình phương A có các tính chất dưới đây:

- Nếu bất kỳ hàng hoặc cột nào của A là zero, thì $\det(A) = 0$.
- Định thức của λ được nhân với A, nếu tất cả các khoản nhập của một hàng bất kỳ (hoặc một cột bất kỳ) của A được nhân với λ .
- Định thức của A sẽ không thay đổi nếu các khoản nhập trong một hàng (cột, theo tương ứng) được bổ sung vào các khoản nhập trong một hàng khác (cột, theo tương ứng).
- Định thức của A bằng định thức của A^T.
- Định thức của A được nhân với -1 nếu hai hàng bất kỳ (hai cột, theo tương ứng) được trao đổi.

Ngoài ra, với mọi ma trận bình phương A và B, ta có $\det(AB) = \det(A) \det(B)$.

Định lý 31.5

Một ma trận $n \times n$ A là lập dị nếu và chỉ nếu $\det(A) = 0$.

Các ma trận xác định dương

Các ma trận xác định dương đóng một vai trò quan trọng trong nhiều

ứng dụng. Một ma trận $n \times n$ A là ***xác định dương*** [positive-definite] nếu $x^T Ax > 0$ với tất cả các vectơ kích cỡ-*n* $x \neq 0$. Ví dụ, ma trận đồng nhất thức là xác định dương, bởi với một vectơ phi zero $x = (x_1\ x_2, \dots x_n)^T$,

$$\begin{aligned} x^T I_n x &= x^T x \\ &= \|x^2\| \\ &= \sum_{i=1}^{n} x_i^2 \\ &> 0 . \end{aligned}$$

Như sẽ thấy, các ma trận nảy sinh trong các ứng dụng thường là xác định dương do định lý dưới đây.

Định lý 31.6

Với bất kỳ ma trận A có hạng cột đầy đủ, ma trận A^TA là xác định dương.

Chứng minh Ta phải chứng tỏ $x^T(A^TA)x > 0$ với bất kỳ vectơ phi zero x. Với bất kỳ vectơ x,

$$\begin{aligned} x^T(A^TA)x &= (Ax)^T(Ax) \quad \text{(theo Bài tập 31.1-3)} \\ &= \|Ax\|^2 \qquad (31.8) \\ &\geq 0 . \end{aligned}$$

Lưu ý, $\|Ax\|^2$ chính là tổng các bình phương của các thành phần của vectơ Ax. Do đó, nếu $\|Ax\|^2 = 0$, mọi thành phần của Ax là 0, tức $Ax = 0$. Bởi A có hạng cột đầy đủ, nên $Ax = 0$ hàm ý $x = 0$, theo Định lý 31.2. Do đó, $A^T A$ là xác định dương.

Các tính chất khác của các ma trận xác định dương sẽ được khảo sát trong Đoạn 31.6.

Bài tập

31.1-1

Chứng minh tích của hai ma trận tam giác dưới chính là tam giác dưới. Chứng minh định thức của một ma trận tam giác (dưới hoặc trên) bằng với tích của các thành phần đường chéo của nó. Chứng minh nghịch đảo của một ma trận tam giác dưới chính là tam giác dưới, nếu nó tồn tại.

31.1-2

Chứng minh nếu P là một ma trận hoán vị $n \times n$ và A là một ma trận $n \times n$, thì PA có thể có được từ A bằng cách hoán vị các hàng của nó, và

AP có thể có được từ A bằng cách hoán vị các cột của nó. Chứng minh tích của hai ma trận hoán vị sẽ là một ma trận hoán vị. Chứng minh nếu P là một ma trận hoán vị, thì P là nghịch đảo, nghịch đảo của nó là P^T, và P^T là một ma trận hoán vị.

31.1-3

Chứng minh $(AB)^T = B^T A^T$ và $A^T A$ luôn là một ma trận đối xứng.

31.1-4

Chứng minh nếu B và C là các nghịch đảo của A, thì $B = C$.

31.1-5

Cho A và B là các ma trận $n \times n$ sao cho $AB = I$. Chứng minh nếu A' có được từ A bằng cách cộng hàng j vào hàng i, thì nghịch đảo B' của A' có thể có được bằng cách trừ cột i với cột j của B.

31.1-6

Cho A là một ma trận phi lập dị $n \times n$ với các khoản nhập số phức. Chứng tỏ mọi khoản nhập của A^{-1} là số thực nếu và chỉ nếu mọi khoản nhập của A là số thực.

31.1-7

Chứng tỏ nếu A là một ma trận đối xứng phi lập dị, thì A^{-1} là đối xứng. Chứng tỏ nếu B là một ma trận (tương thích) tùy ý, thì BAB^T là đối xứng.

31.1-8

Chứng tỏ một ma trận A có hạng cột đầy đủ nếu và chỉ nếu $Ax = 0$ hàm ý $x = 0$. (*Mách nước:* Biểu diễn sự phụ thuộc tuyến tính của một cột trên các cột khác dưới dạng một phương trình ma trận-vectơ.)

31.1-9

Chứng minh với bất kỳ hai ma trận tương thích A và B,

$$\text{rank}(AB) \le \min(\text{rank}(A), \text{rank}(B)),$$

ở đó đẳng thức đứng vững nếu A hoặc B là một ma trận bình phương phi lập dị. (*Mách nước.* Dùng định nghĩa thay thế về hạng của một ma trận.)

31.1-10

Cho các số $x_0, x_1, \ldots, x_{n-1}$, chứng minh định thức của ***ma trận Vandermonde***

$$V(x_0, x_1, \ldots, x_{n-1}) = \begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix}$$

is

$$(V(x_0, x_1, \ldots, x_{n-1})) = \prod_{0 \le j \le k \le n-1} (x_k - x_j) .$$

(*Mách nước:* Nhân cột i với $-x_0$ và bổ sung nó vào cột $i + 1$ với $i = n - 1, n - 2, \ldots, 1$, rồi dùng phương pháp quy nạp.)

31.2 Thuật toán Strassen với phép nhân ma trận

Đoạn này trình bày thuật toán đệ quy khác thường của Strassen để nhân các ma trận $n \times n$ chạy trong $\Theta(n^{\lg 7}) = O(n^{2.81})$ thời gian. Do đó, với n, đủ lớn, nó làm tốt hơn thuật toán phép nhân ma trận $\Theta(n^3)$ đơn sơ MATRIX-MULTIPLY trong Đoạn 26.1.

Khái quát về thuật toán

Có thể xem thuật toán Strassen như là một ứng dụng của một kỹ thuật thiết kế quen thuộc: chia để trị. Giả sử ta muốn tính toán tích $C = AB$, ở đó mỗi trong số A, B, và C là các ma trận $n \times n$. Giả sử n là một lũy thừa chính xác của 2, ta chia mỗi trong số A, B, và C thành bốn ma trận $n/2 \times n/2$, viết lại phương trình $C = AB$ như sau:

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & g \\ f & h \end{pmatrix} . \qquad (31.9)$$

(Bài tập 31.2-2 đối phó với tình huống ở đó n không phải là một lũy thừa chính xác của 2.) Để tiện dụng, các ma trận con của A được gán nhãn theo thứ tự abc từ trái qua phải, trong khi đó các ma trận con của B được gán nhãn theo thứ tự abc từ trên xuống, phù hợp với cách thực hiện phép nhân ma trận. Phương trình (31.9) tương ứng với bốn phương trình

$$r = ae + bf , \qquad (31.10)$$

$$s = ag + bh , \qquad (31.11)$$

$$t = ce + df , \qquad (31.12)$$

$$u = cg + dh . \qquad (31.13)$$

Mỗi trong số bốn phương trình này chỉ định hai phép nhân của các ma trận $n/2 \times n/2$ và phép cộng của các tích $n/2 \times n/2$ của chúng. Dùng các phương trình này để định nghĩa một chiến lược chia để trị dễ hiểu, ta suy ra phép truy toán dưới đây trong thời gian $T(n)$ để nhân hai ma trận $n \times n$:

$$T(n) = 8T(n/2) + \Theta(n^2)\ . \qquad (31.14)$$

Đáng tiếc, phép truy toán (31.14) có nghiệm $T(n) = \Theta(n^3)$, và như vậy phương pháp này không nhanh hơn phương pháp bình thường.

Strassen đã khám phá một cách tiếp cận đệ quy khác chỉ yêu cầu 7 phép nhân đệ quy của các ma trận $n/2 \times n/2$ và $\Theta(n^2)$ phép cộng và phép trừ vô hướng, cho ra phép truy toán

$$\begin{aligned} T(n) &= 7T(n/2) + \Theta(n^2) \qquad (31.15)\\ &= \Theta(n^{\lg 7}) \\ &= O(n^{2.81})\ . \end{aligned}$$

Phương pháp Strassen có bốn bước:

1. Chia các ma trận đầu vào A và B thành các ma trận con $n/2 \times n/2$, như trong phương trình (31.9).

2. Dùng $\Theta(n^2)$ phép cộng và phép trừ vôhướng, tính toán 14 ma trận $n/2 \times n/2$ $A_1, B_1, A_2, B_2, \ldots, A_7, B_7$.

3. Theo đệ quy tính toán bảy tích ma trận $P_i = A_iB_i$ với $i = 1, 2, \ldots, 7$.

4. Tính toán các ma trận con mong muốn r, s, t, u của kết quả ma trận C bằng cách cộng và/hoặc trừ các tổ hợp khác nhau của các ma trận P_i, chỉ dùng $\Theta(n^2)$ phép cộng và phép trừ vô hướng.

Một thủ tục như vậy thỏa phép truy toán (31.15). Giờ đây tất cả những gì phải làm đó là điền vào các chi tiết thiếu.

Xác định các tích ma trận con

Ta không biết rõ chính xác cách thức mà Strassen đã khám phá các tích ma trận con là chìa khóa để khiến thuật toán của ông làm việc. Ở đây, ta kiến tạo lại một phương pháp khám phá đáng tin cậy.

Ta hãy suy đoán rằng mỗi tích ma trận P_i có thể được viết theo dạng

$$\begin{aligned} P_i &= A_iB_i \\ &= (\alpha_{i1}a + \alpha_{i2}b + \alpha_{i3}c + \alpha_{i4}d) \cdot (\beta_{i1}e + \beta_{i2}f + \beta_{i3}g + \beta_{i4}h)\ , \qquad (31.16) \end{aligned}$$

ở đó tất cả các hệ số α_{ij}, β_{ij} được rút từ tập hợp $\{-1, 0, 1\}$. Nghĩa là, ta suy đoán rằng mỗi tích được tính toán bằng cách cộng hoặc trừ vài ma trận con của A, cộng hoặc trừ vài ma trận con của B, rồi nhân hai kết quả

với nhau. Tuy có thể áp dụng các chiến lược chung hơn, song chiến lược đơn giản này sẽ làm việc.

Nếu ta lập thành tất cả các tích của chúng ta theo cách này, thì ta có thể dùng phương pháp này một cách đệ quy mà không cần mặc nhận tính giao hoán của phép nhân, bởi mỗi tích có tất cả các ma trận con A bên trái và tất cả các ma trận con B bên phải. Tính chất này là thiết yếu cho ứng dụng đệ quy của phương pháp này, bởi phép nhân ma trận không giao hoán.

Để tiện dụng, ta sẽ dùng các ma trận 4 x 4 để biểu diễn các tổ hợp tuyến tính của các tích của các ma trận con, ở đó mỗi tích tổ hợp một ma trận con của A với một ma trân con của B như trong phương trình (31.16). Ví dụ, ta có thể viết lại phương trình (31.10) là

$$
\begin{aligned}
r &= ae + bf \\
&= (a\ b\ c\ d) \begin{pmatrix} +1 & 0 & 0 & 0 \\ 0 & +1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} e \\ f \\ g \\ h \end{pmatrix} \\
&= \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} e & f & g & h \end{array} \\ \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \end{array}
\end{aligned}
$$

Biểu thức cuối sử dụng một hệ ký hiệu viết tắt ở đó "+" biểu diễn +l, "." biểu diễn 0, và "–" biểu diễn −1. (Từ đây trở đi, ta bỏ qua các nhãn hàng và cột.) Dùng hệ ký hiệu này, ta có các phương trình dưới đây cho các ma trận con khác của ma trận kết quả C :

$$
\begin{aligned}
s &= ag + bh \\
&= \begin{pmatrix} \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} , \\
t &= ce + df \\
&= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \end{pmatrix} \\
u &= cg + dh
\end{aligned}
$$

$$= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix}.$$

Ta bắt đầu đợt tìm kiếm một thuật toán ma trận-phép nhân nhanh hơn bằng cách nhận xét rằng ma trận con s có thể được tính toán là $s = P_1 + P_2$, ở đó P_1 và P_2 được tính toán bằng một phép nhân ma trận mỗi:

$$\begin{aligned} P_1 &= A_1B_1 \\ &= a \cdot (g - h) \\ &= ag - ah \\ &= \begin{pmatrix} \cdot & \cdot & + & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}, \end{aligned}$$

$$\begin{aligned} P_2 &= A_2B_2 \\ &= (a + b) \cdot h \\ &= ah + bh \\ &= \begin{pmatrix} \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}. \end{aligned}$$

Ma trận t có thể được tính toán theo cách tương tự là $t = P_3 + P_4$, ở đó

$$\begin{aligned} P_3 &= A_3B_3 \\ &= (c + d) \cdot e \\ &= ce + de \\ &= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \end{pmatrix}, \end{aligned}$$

và

$$\begin{aligned} P_4 &= A_4B_4 \\ &= d \cdot (f - e) \\ &= df - de \\ &= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & + & \cdot & \cdot \end{pmatrix}. \end{aligned}$$

Ta hãy định nghĩa một ***số hạng cốt yếu*** là một trong số tám số hạng xuất hiện phía bên phải của một trong các phương trình (31.10)-(31.13). Giờ đây, ta đã dùng 4 tích để tính toán hai ma trận con s và t có các số hạng cốt yếu là ag, bh, ce, và df. Lưu ý, P_1 tính toán số hạng cốt yếu ag, P_2 tính toán số hạng cốt yếu bh, P_3 tính toán số hạng cốt yếu ce, và P_4 tính toán số hạng cốt yếu df. Như vậy, ta chỉ cần tính toán hai ma trận con còn lại r và u, có các số hạng cốt yếu là các số hạng đường chéo ae, bf, cg, và dh, mà không dùng trên 3 tích bổ sung. Giờ đây ta thử sự cách tân P_5 để tính toán hai số hạng cốt yếu cùng một lúc:

$$\begin{aligned}
P_5 &= A_5B_5 \\
&= (a+d).(e+h) \\
&= ae + ah + de + dh \\
&= \begin{pmatrix} + & . & . & + \\ . & . & . & . \\ . & . & . & . \\ + & . & . & + \end{pmatrix}.
\end{aligned}$$

Ngoài việc tính toán cả hai số hạng cốt yếu ae và dh, P_5 tính toán các số hạng không cốt yếu ah và de, cần được khử bằng cách nào đó. Ta có thể dùng P_4 và P_2 để khử chúng, nhưng như vậy hai số hạng không cốt yếu kia xuất hiện:

$$\begin{aligned}
P_5 + P_4 - P_2 &= ae + dh + df - bh \\
&= \begin{pmatrix} + & . & . & . \\ . & . & . & - \\ . & . & . & . \\ . & + & . & + \end{pmatrix}.
\end{aligned}$$

Nhờ cộng một tích bổ sung

$$\begin{aligned}
P_6 &= A_6B_6 \\
&= (b-d).(f+h) \\
&= bf + bh - df - dh \\
&= \begin{pmatrix} . & . & . & . \\ . & + & . & + \\ . & . & . & . \\ . & - & . & - \end{pmatrix},
\end{aligned}$$

tuy nhiên, ta được

$$\begin{aligned}
r &= P_5 + P_4 - P_2 + P_6 \\
&= ae + bf
\end{aligned}$$

$$= \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} .$$

Ta có thể được u theo cách tương tự từ P_5 bằng cách dùng P_1 và P_3 để dời các số hạng không cốt yếu của P_5 theo một hướng khác:

$$P_5 + P_1 - P_3 = ae + ag - ce + dh$$
$$= \begin{pmatrix} + & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix} .$$

Bằng cách trừ một tích bổ sung

$$\begin{aligned} P_7 &= A_7 B_7 \\ &= (a - c) \cdot (e + g) \\ &= ae + ag - ce - cg \\ &= \begin{pmatrix} + & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & - & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} , \end{aligned}$$

giờ đây ta được

$$\begin{aligned} u &= P_5 + P_1 - P_3 - P_7 \\ &= cg + dh \\ &= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix} . \end{aligned}$$

Như vậy, 7 tích ma trận con $P_1, P_2, \ldots, P_7$ có thể được dùng để tính toán tích $C = AB$, hoàn tất phần mô tả của phương pháp Strassen.

Thảo luận

Sự bất biến lớn ẩn trong thời gian thực hiện của thuật toán Strassen khiến nó không thực tiễn trừ phi các ma trận là lớn (n ít nhất là khoảng 45) và trù mật (ít các khoản nhập zero). Với các ma trận nhỏ, thuật toán đơn giản thường được ưa dùng, và với các ma trận thưa, lớn, trong thực tế ta có các thuật toán ma trận thưa đặc biệt tốt hơn thuật toán Strassen. Như vậy, phương pháp của Strassen phần lớn chỉ được quan tâm về mặt lý thuyết. ●

Nhờ dùng các kỹ thuật cao cấp vượt quá phạm vi của tài liệu này, ta có thể thực tế nhân các ma trận $n \times n$ trong thời gian tốt hơn $\Theta(n^{\lg 7})$. Cận

trên tốt nhất hiện hành xấp xỉ khoảng $O(n^{2.376})$. Cận dưới tốt nhất được biết chính là cận hiển nhiên $\Omega(n^2)$ (hiển nhiên bởi vì ta phải điền vào n^2 thành phần của tích ma trận). Như vậy, hiện ta không biết phép nhân ma trận thực sự khó đến mức nào.

Thuật toán Strassen không yêu cầu các khoản nhập ma trận là các số thực. Tất cả vấn đề đó là hệ thống số hình thành một vành đại số. Tuy nhiên, nếu các khoản nhập ma trận không hình thành một vành, đôi lúc các kỹ thuật khác có thể được vận dụng để cho phép áp dụng phương pháp của ông. Các vấn đề này được mô tả đầy đủ hơn trong đoạn kế tiếp.

Bài tập

31.2-1

Dùng thuật toán Strassen để tính toán tích ma trận

$$\begin{pmatrix} 1 & 3 \\ 5 & 7 \end{pmatrix}\begin{pmatrix} 8 & 4 \\ 6 & 2 \end{pmatrix} .$$

Nêu công của bạn.

31.2-2

Nêu cách sửa đổi thuật toán Strassen của bạn để nhân các ma trận $n \times n$ ở đó n không phải là một lũy thừa chính xác của 2? Chứng tỏ thuật toán kết quả chạy trong thời gian $\Theta(n^{\lg 7})$.

31.2-3

Đâu là k lớn nhất sao cho nếu có thể nhân các ma trận 3×3 dùng k phép nhân (không mặc nhận tính giao hoán của phép nhân), thì bạn có thể nhân các ma trận $n \times n$ trong thời gian $\Theta(n^{\lg 7})$? Đâu là thời gian thực hiện của thuật toán này?

31.2-4

V. Pan đã khám phá ra một cách nhân các ma trận 68×68 dùng 132,464 phép nhân, một cách nhân các ma trận 70×70 dùng 143,640 phép nhân, và một cách nhân các ma trận 72×72 dùng 155,424 phép nhân. Phương pháp nào cho ra thời gian thực hiện tiệm cận tốt nhất khi được dùng trong một thuật toán ma trận-phép nhân chia để trị? So sánh nó với thời gian thực hiện của thuật toán Strassen.

31.2-5

Có thể nhân một ma trận $kn \times n$ với một ma trận $n \times kn$ nhanh tới mức nào, dùng thuật toán Strassen làm một chương trình con? Trả lời

cùng câu hỏi với thứ tự của các ma trận đầu vào đảo ngược.

31.2-6

Nêu cách nhân các số phức $a + bi$ và $c + di$ chỉ dùng ba phép nhân số thực. Thuật toán sẽ nhận a, b, c, và d làm đầu vào và tạo ra thành phần thực $ac - bd$ và thành phần tưởng tượng $ad + bc$ một cách tách biệt.

---

* 31.3 Các hệ thống số đại số và phép nhân ma trận bool

Các tính chất của phép cộng và phép nhân ma trận tùy thuộc vào các tính chất của hệ thống số cơ sở. Trong đoạn này, ta định nghĩa ba kiểu hệ thống số cơ sở khác nhau: các tựa vành, các vành, và các trường.

Ta có thể định nghĩa phép nhân ma trận trên các tựa vành [quasirings], và thuật toán ma trận phép nhân Strassen làm việc trên các vành. Sau đó, ta trình bày một thủ thuật đơn giản để rút gọn phép nhân ma trận bool, được định nghĩa trên một tựa vành không phải là một vành, thành phép nhân trên một vành. Cuối cùng, ta đề cập tại sao các tính chất của một trường không thể khai thác một cách tự nhiên để cung cấp các thuật toán tốt hơn cho phép nhân ma trận.

Các tựa vành

Cho $(S, \oplus, \odot, \bar{0}, \bar{1})$ thể hiện một hệ thống số, ở đó S là một tập hợp các thành phần, $\oplus$ và $\odot$ là các phép toán nhị phân trên S (phép cộng và phép nhân, theo thứ tự nêu trên), và $\bar{0}$ và $\bar{1}$ là các thành phần được đánh dấu riêng biệt của S.

Hệ thống này là một ***tựa vành*** nếu nó thỏa các tính chất dưới đây:

1. $(S, \oplus, \bar{0})$ là một ***nửa nhóm*** [monoid]:

- S là ***đóng*** [closed] dưới $\oplus$; nghĩa là, $a \oplus b \in S$ với tất cả $a, b \in S$.
- $\oplus$ là ***kết hợp*** [associative]; nghĩa là, $a \oplus (b \oplus c) = (a \oplus b) \oplus c$ với tất cả $a, b, c \in S$.
- $\bar{0}$ là một ***đồng nhất thức*** với $\oplus$; nghĩa là, $a \oplus \bar{0} = \bar{0} \oplus a = a$ với tất cả $a \in S$.

Cũng vậy, $(S, \odot, \bar{1})$ là một nửa nhóm.

2. $\bar{0}$ là một ***phần tử rỗng*** [annihilator]; nghĩa là, $a \odot \bar{0} = \bar{0} \odot a = \bar{0}$ với tất cả $a \in S$.

3. Toán tử $\oplus$ là ***giao hoán***; nghĩa là, $a \oplus b = b \oplus a$ với tất cả $a, b \in S$.

4. Toán tử $\odot$ ***phân phối*** trên $\oplus$; nghĩa là, $a \odot (b \oplus c) = (a \odot b) \oplus (a \odot c)$ và $(b \oplus c) \odot a = (b \odot a) \oplus (c \odot a)$ với tất cả $a, b, c \in S$.

Các ví dụ về các tựa vành bao gồm ***tựa vành bool*** $(\{0,1\}, \vee, \wedge, 0, 1)$, ở đó $\vee$ thể hiện OR lôgic và $\wedge$ thể hiện AND lôgic, và hệ thống số tự nhiên $(\mathbf{N}, +, \cdot, 0, 1)$, ở đó $+$ và $\cdot$ thể hiện phép cộng và phép nhân bình thường. Mọi tựa vành đóng (xem Đoạn 26.4) cũng là một tựa vành: các tựa vành đóng tuân thủ tính lũy đẳng bổ sung và các tính chất tổng-vô hạn.

Ta có thể mở rộng $\oplus$ và $\odot$ sang các ma trận như đã làm với $+$ và $\cdot$ trong Đoạn 31.1. Thể hiện ma trận đồng nhất thức $n \times n$ bao gồm $\bar{0}$ và $\bar{1}$ bằng $\bar{I}_n$, ta thấy rằng phép nhân ma trận được định nghĩa kỹ và bản thân ma trận hệ thống là một tựa vành, như định lý dưới đây phát biểu.

Định lý 31.7 (Các ma trận trên một tựa vành hình thành một tựa vành)

Nếu $(S, \oplus, \odot, \bar{0}, \bar{1})$ là một tựa vành và $n \geq 1$, thì $(S^{n \times n}, \oplus, \odot, \bar{0}, \bar{I}_n)$ là một tựa vành.

Chứng minh Phần chứng minh được chừa để làm Bài tập 31.3-3.

Các vành

Phép trừ không được định nghĩa cho các tựa vành, nhưng được định nghĩa cho một ***vành*** [ring], là một tựa vành $(S, \oplus, \odot, \bar{0}, \bar{1})$ thỏa tính chất bổ sung dưới đây:

5. Mọi thành phần trong S có một ***nghịch đảo cộng tính***; nghĩa là, với tất cả $a \in S$, ở đó tồn tại một thành phần $b \in S$ sao cho $a \oplus b = b \oplus a = \bar{0}$. Một b như vậy còn gọi là ***âm*** của a và được ký hiệu là $(-a)$.

Cho âm của bất kỳ thành phần nào được định nghĩa, ta có thể định nghĩa phép trừ bằng $a - b = a + (-b)$.

Có nhiều ví dụ về các vành. Các số nguyên $(\mathbf{Z}, +, \cdot, 0, 1)$ dưới các phép toán cộng và nhân bình thường sẽ hình thành một vành. Các số nguyên modulo n với bất kỳ số nguyên $n > 1$—nghĩa là, $(\mathbf{Z}_n, +, \cdot, 0, 1)$, ở đó $+$ là phép cộng modulo n và $\cdot$ là phép nhân modulo n—hình thành một vành. Một ví dụ khác đó là tập hợp $\mathbf{R}[x]$ các đa thức bậc hữu hạn trong x với các hệ số thực dưới các phép toán bình thường—nghĩa là, $(\mathbf{R}[x], +, \cdot, 0, 1)$, ở đó $+$ là phép cộng đa thức và $\cdot$ là phép nhân đa thức.

Hệ luận dưới đây chứng tỏ Định lý 31.7 tổng quát hóa thành các vành một cách tự nhiên.

Hệ luận 31.8 (Các ma trận trên một vành hình thành một vành)

Nếu $(S, \oplus, \odot, \bar{0}, \bar{1})$ là một vành và $n \geq 1$, thì $(S^{n \times n}, \oplus, \odot, \bar{0}, \bar{I}_n)$ là một vành.

Chứng minh Chứng minh được để lại làm Bài tập 31.3-3.

Dùng hệ luận này, ta có thể chứng minh định lý dưới đây.

Định lý 31.9

Thuật toán ma trận-phép nhân của Strassen làm việc đúng đắn trên mọi vành các thành phần ma trận.

Chứng minh Thuật toán Strassen tùy thuộc vào tính đúng đắn của thuật toán dành cho các ma trận 2 x 2, chỉ yêu cầu các thành phần ma trận thuộc về một vành. Bởi các thành phần ma trận thuộc về một vành, nên Hệ luận 31.8 hàm ý chính các ma trận hình thành một vành. Như vậy, theo phương pháp quy nạp, thuật toán Strassen làm việc đúng đắn tại mỗi cấp đệ quy.

Thực tế, thuật toán Strassen dành cho phép nhân ma trận tùy thuộc nhiều vào sự tồn tại của các nghịch đảo cộng tính. Trong số bảy tích P_1, $P_2, \ldots, P_7$, bốn tích kéo theo những khác biệt của các ma trận con. Như vậy, nói chung thuật toán Strassen không làm việc cho các tựa vành.

Phép nhân ma trận bool

Không thể trực tiếp dùng thuật toán Strassen để nhân các ma trận bool, bởi tựa vành bool $(\{0, 1\}, \vee, \wedge, 0, 1)$ không phải là một vành. Có những trường hợp ở đó một tựa vành được chứa trong một hệ thống lớn hơn mà hệ thống này là một vành. Ví dụ, các số tự nhiên (một tựa vành) là một tập hợp con của các số nguyên (một vành), và do đó thuật toán Strassen có thể được dùng để nhân các ma trận các số tự nhiên nếu ta xét hệ thống số cơ sở là các số nguyên. Đáng tiếc, tựa vành bool không thể mở rộng theo cách tương tự như một vành. (Xem Bài tập 31.3-4.)

Định lý dưới đây trình bày một thủ thuật đơn giản để rút gọn phép nhân ma trận bool thành phép nhân trên một vành. Bài toán 31-1 trình bày một cách tiếp cận hiệu quả khác.

Định lý 31.10

Nếu $M(n)$ thể hiện số lượng các phép toán số học cần có để nhân hai ma trận $n \times n$ trên các số nguyên, thì hai ma trận bool $n \times n$ có thể được nhân bằng $O(M(n))$ phép toán số học.

Chứng minh Cho hai ma trận là A và B, và cho $C = AB$ trong tựa vành bool, nghĩa là,

$$c_{ij} = \bigvee_{k=1}^{n} a_{ik} \wedge b_{kj} .$$

Thay vì tính toán trên tựa vành bool, ta tính toán tích C' trên vành các số nguyên bằng thuật toán ma trận-phép nhân đã cho sử dụng $M(n)$ phép toán số học. Như vậy ta có

$$c'_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} .$$

Mỗi số hạng $a_{ik}b_{kj}$ của tổng này là 0 nếu và chỉ nếu $a_{ik} \wedge b_{kj} = 0$, và 1 nếu và chỉ nếu $a_{ik} \wedge b_{kj} = 1$. Như vậy, tổng số nguyên c'_{ij} là 0 nếu và chỉ nếu mọi số hạng là 0 hoặc, theo tương đương, nếu và chỉ nếu OR bool của các số hạng, là c_{ij}, là 0. Do đó, ma trận bool C có thể được tái thiết với $\Theta(n^2)$ phép toán số học từ ma trận số nguyên C' bằng cách đơn giản so sánh từng c'_{ij} với 0. Như vậy, số lượng phép toán số học cho nguyên cả thủ tục là $O(M(n)) + \Theta(n^2) = O(M(n))$, bởi $M(n) = \Omega(n^2)$.

Như vậy, dùng thuật toán Strassen, ta có thể thực hiện phép nhân ma trận bool trong $O(n^{\lg 7})$ thời gian.

Phương pháp bình thường nhân các ma trận bool chỉ sử dụng các biến bool. Tuy nhiên, nếu ta dùng kiểu thích ứng này của thuật toán Strassen, tích ma trận chung cuộc có thể có các khoản nhập lớn bằng n, như vậy đòi hỏi một từ máy tính để lưu trữ chúng thay vì một bit đơn. Rắc rối hơn đó là các kết quả trung gian, là các số nguyên, có thể tăng trưởng thậm chí lớn hơn. Một phương pháp để giữ cho các kết quả trung gian không tăng trưởng quá lớn đó là thực hiện tất cả các phép tính modulo $n+1$. Bài tập 31.3-5 yêu cầu bạn chứng tỏ modulo $n+1$ làm việc không ảnh hưởng đến tính đúng đắn của thuật toán.

Các trường

Một vành $(S, \oplus, \odot, \bar{0}, \bar{1})$ là một ***trường*** [field] nếu nó thỏa hai tính chất bổ sung dưới đây:

6. Toán tử $\odot$ là ***giao hoán***; nghĩa là, $a \odot b = b \odot a$ với tất cả $a, b \in S$.

7. Mọi thành phần phi zero trong S có một ***nghịch đảo nhân***; nghĩa là, với tất cả

$a \in S - \{\bar{0}\}$, ở đó tồn tại một thành phần $b \in S$ sao cho $a \odot b = b \odot a = \bar{1}$. Một thành phần b như vậy thường được gọi là ***nghịch đảo*** của a và được ký hiệu là a^{-1}.

Các ví dụ về các trường bao gồm các số thực $(\mathbf{R}, +, \cdot, 0, 1)$, các số phức $(\mathbf{C}, +, \cdot, 0, 1)$, và các số nguyên modulo một số nguyên tố p: $(\mathbf{Z}_p, +, \cdot, 0, 1)$.

Bởi các trường cung cấp các nghịch đảo nhân của các thành phần,

nên phép chia là khả dĩ. Chúng cũng cung cấp tính giao hoán. Nhờ tổng quát hóa từ các tựa vành thành các vành, Strassen có thể cải thiện thời gian thực hiện của phép nhân ma trận. Bởi các thành phần cơ sở của các ma trận thường xuất xứ từ một trường—ví dụ, các số thực—nên ta có thể hy vọng rằng nhờ dùng các trường thay vì các vành trong một thuật toán đệ quy tương tự Strassen, thời gian thực hiện có thể được cải thiện thêm.

Cách tiếp cận này dường như không chắc mang lại lợi ích. Với một thuật toán đệ quy chia để trị dựa trên các trường để làm việc, các ma trận tại mỗi bước đệ quy phải hình thành một trường. Đáng tiếc, phần mở rộng tự nhiên của Định lý 31.7 và Hệ luận 31.8 ra các trường đều thất bại trầm trọng. Với $n > 1$, tập hợp các ma trận $n \times n$ *không bao giờ* hình thành một trường, cho dù hệ thống số cơ sở là một trường. Phép nhân các ma trận $n \times n$ không giao hoán, và nhiều ma trận $n \times n$ không có các nghịch đảo. Do đó, các thuật toán tốt hơn cho phép nhân ma trận thường dựa trên lý thuyết vành nhiều hơn lý thuyết trường.

Bài tập

31.3-1 *

Thuật toán Strassen có làm việc trên hệ thống số $(\mathbf{Z}[x], +, \cdot, 0, 1)$, ở đó $\mathbf{Z}[x]$ là tập hợp tất cả các đa thức với các hệ số số nguyên trong biến x và $+$ và $\cdot$ là phép cộng và phép nhân đa thức bình thường không?

31.3-2 *

Giải thích tại sao thuật toán Strassen không làm việc trên các tựa vành đóng (xem Đoạn 26.4) hoặc trên tựa vành bool $(\{0, 1\}, \vee, \wedge, 0, 1)$.

31.3-3 *

Chứng minh Định lý 31.7 và Hệ luận 31.8.

31.3-4 *

Chứng tỏ tựa vành bool $(\{0, 1\}, \vee, \wedge, 0, 1)$ không thể nhúng trong một vành. Nghĩa là, chứng tỏ không thể bổ sung một "-1" vào tựa vành sao cho cấu trúc đại số kết quả là một vành.

31.3-5

Chứng tỏ nếu tất cả các phép tính trong thuật toán của Định lý 31.10 được thực hiện modulo $n + 1$, thuật toán vẫn làm việc đúng đắn.

31.3-6

Nêu cách xác định một cách hiệu quả nếu một đồ thị đầu vào không

hướng đã cho chứa một tam giác (một tập hợp ba đỉnh kề qua lại).

31.3-7 *

Chứng tỏ việc tính toán tích của hai ma trận bool $n \times n$ trên tựa vành bool có thể rút gọn thành tiến trình tính toán bao đóng bắc cầu [transitive closure] của một đồ thị đầu vào có hướng $3n$-đỉnh đã cho.

31.3-8

Nêu cách tính toán bao đóng bắc cầu của một đồ thị đầu vào có hướng n-đỉnh đã cho trong thời gian $O(n^{\lg 7} \lg n)$. So sánh kết quả này với khả năng thực hiện của thủ tục TRANSITIVE-CLOSURE trong Đoạn 26.2.

31.4 Giải các hệ thống phương trình tuyến tính

Việc giải một tập hợp các phương trình tuyến tính tương tự là một bài toán căn bản xảy ra trong các ứng dụng đa dạng. Có thể diễn tả một hệ thống tuyến tính như một phương trình ma trận ở đó mỗi thành phần ma trận hoặc vectơ thuộc về một trường, thường là các số thực **R**. Đoạn này mô tả cách giải một hệ thống các phương trình tuyến tính dùng một phương pháp có tên phân tích LUP [LUP decomposition].

Ta bắt đầu bằng một tập hợp các phương trình tuyến tính trong n ẩn số $x_1, x_2, \ldots, x_n$

$$
\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 \\
&\vdots \\
a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n &= b_n .
\end{aligned}
\tag{31.17}
$$

Một tập hợp các giá trị với $x_1, x_2, \ldots, x_n$ thỏa tất cả các phương trình (31.17) đồng thời được gọi là một ***nghiệm*** cho các phương trình này. Trong đoạn này, ta chỉ xem xét trường hợp ở đó có chính xác n phương trình trong n ẩn số.

Để tiện dụng, ta có thể viết lại các phương trình (31.17) dưới dạng phương trình ma trận-vectơ

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

hoặc, theo tương đương, cho $A = (a_{ij})$, $x = (x_j)$, và $b = (b_i)$, là

$$Ax = b. \tag{31.18}$$

Nếu A là phi lập dị, nó có một nghịch đảo A^{-1}, và

$$x = A^{-1}b \tag{31.19}$$

là vectơ nghiệm. Ta có thể chứng minh x là nghiệm duy nhất cho phương trình (31.18) như sau. Nếu có hai giải pháp, x và x', thì $Ax = Ax' = b$ và

$$\begin{aligned} x &= (A^{-1}A)x \\ &= A^{-1}(Ax) \\ &= A^{-1}(Ax') \\ &= (A^{-1}, A)x' \\ &= x' . \end{aligned}$$

Trong đoạn này, ta sẽ chủ yếu tập trung vào trường hợp ở đó A là phi lập dị hoặc, theo tương đương (theo Định lý 31.1), hạng của A bằng với số n ẩn số. Tuy nhiên, có các khả năng khác mà ta cũng nên đề cập ngắn gọn. Nếu số lượng các phương trình nhỏ hơn số n ẩn số—hoặc, chung hơn, nếu hạng của A nhỏ hơn n—thì hệ thống được ***xác định dưới*** [underdetermined]. Một hệ thống xác định dưới thường có nhiều nghiệm vô số kể (xem Bài tập 31.4-9), mặc dù nó có thể không có nghiệm nào cả nếu các phương trình không nhất quán. Nếu số lượng các phương trình vượt quá số n ẩn số, hệ thống được ***xác định trên*** [overdetermined], và có thể không tồn tại bất kỳ nghiệm nào. Việc tìm các nghiệm xấp xỉ tốt cho các hệ thống xác định trên của các phương trình tuyến tính là một bài toán quan trọng sẽ được đề cập trong Đoạn 31.6.

Ta hãy quay về bài toán giải hệ thống $Ax = b$ của n phương trình trong n ẩn số. Một cách tiếp cận đó là tính toán A^{-1} rồi nhân cả hai cạnh với A^{-1}, cho ra $A^{-1}Ax = A^{-1}b$, hoặc $x = A^{-1}b$. Trong thực tế, cách tiếp cận này phải chịu ***tính không ổn định về số***: các lỗi làm tròn có khuynh hướng tích lũy không chính đáng khi các phép biểu diễn số chấm bậcng được dùng thay vì các số thực lý tưởng. May thay, có một cách tiếp cận khác—phân tích LUP—ổn định về số và có ưu điểm nhanh hơn khoảng một thừa số của 3.

Khái quát về phân tích LUP

Ý tưởng đàng sau phân tích LUP đó là tìm ba ma trận $n \times n$ L, U, và P sao cho

$$PA = LU, \qquad (31.20)$$

ở đó

- L là một ma trận tam giác dưới đơn vị,
- U là một ma trận tam giác trên, và
- P là một ma trận hoán vị.

Ta gọi các ma trận L, U, và P thỏa phương trình (31.20) là một ***phân tích LUP*** của ma trận A. Ta sẽ chứng tỏ mọi ma trận phi lập dị A có một phân tích như vậy.

Ưu điểm của việc tính toán một phân tích LUP cho ma trận A đó là có thể giải các hệ thống tuyến tính một cách sẵn sàng hơn khi chúng là tam giác, như trường hợp cả hai ma trận L và U. Sau khi tìm thấy một phân tích LUP cho A, ta có thể giải phương trình (31.18) $Ax = b$ bằng cách chỉ giải các hệ thống tuyến tính tam giác, như sau. Nhân cả hai cạnh của $Ax = b$ với P sẽ cho ra phương trình tương đương $PAx = Pb$, mà theo Bài tập 31.1-2 chung quy là hoán vị các phương trình (31.17). Dùng phân tích của chúng ta (31.20), ta được

$$LUx = Pb.$$

Giờ đây, ta có thể giải phương trình này bằng cách giải hai hệ thống tuyến tính tam giác. Ta hãy định nghĩa $y = Ux$, ở đó x là vectơ nghiệm mong muốn. Trước tiên, ta giải hệ thống tam giác dưới

$$Ly = Pb \qquad (31.21)$$

với vectơ chưa biết y bằng một phương pháp có tên "phép thay tới" [forward substitution]. Sau khi giải đối với y, ta giải hệ thống tam giác trên

$$Ux = y \qquad (31.22)$$

với x chưa biết bằng một phương pháp có tên "phép thay lui" [back substitution]. Vectơ x là giải pháp cho $Ax = b$, bởi ma trận hoán vị P là nghịch đảo (Bài tập 31.1-2):

$$\begin{aligned} Ax &= P^{-1}LUx \\ &= P^{-1}Ly \\ &= P^{-1}Pb \\ &= b \end{aligned}$$

Bước kế tiếp đó là nêu cách làm việc của phép thay tới và phép thay lui sau đó giải bài toán tính toán chính bản thân phân tích LUP.

Phép thay tới và lui

Phép thay tới có thể giải hệ thống tam giác dưới (31.21) trong $\Theta(n^2)$ thời gian, căn cứ vào L, P, và b. Để tiện dụng, ta biểu diễn súc tích phép hoán vị P bằng một mảng $\pi[1..n]$. Với $i = 1, 2, ..., n$, khoản nhập $\pi[i]$ nêu rõ $P_{i,\pi[i]} = 1$ và $P_{ij} = 0$ với $j \neq \pi[i]$. Như vậy, PA có $a_{\pi[i],j}$ trong hàng i và cột j, và Pb có $b_{\pi[i]}$ làm thành phần thứ i của nó. Bởi L là tam giác thấp hơn đơn vị, phương trình (31.21) có thể được viết lại là

$$\begin{array}{lllllll} y_1 & & & & & = & b_{\pi[1]}, \\ l_{21}y_1 & + & y_2 & & & = & b_{\pi[2]}, \\ l_{31}y_1 & + & l_{32}y_2 & + & y_3 & = & b_{\pi[3]}, \\ \vdots & & & & & & \\ l_{n1}y_1 & + & l_{n2}y_2 & + & l_{n3}y_3 + \dots + y_n & = & b_{\pi[n]}. \end{array}$$

Khá hiển nhiên, ta có thể giải trực tiếp cho y_1, bởi phương trình đầu tiên cho biết $y_1 = b_{\pi[1]}$. Sau khi giải với y_1, ta có thể thay nó vào phương trình thứ hai, cho ra

$$y_2 = b_{\pi[2]} - l_{21}y_1 .$$

Giờ đây, ta có thể thay cả y_1 lẫn y_2 vào phương trình thứ ba, để có

$$y_3 = b_{\pi[3]} - (l_{31}y_1 + l_{32}y_2) .$$

Nói chung, ta thay $y_1, y_2, ..., y_{i-1}$ "tới" vào phương trình thứ i để giải với y_i:

$$y_i = b_{\pi[i]} - \sum_{j=1}^{i-1} l_{ij}y_j .$$

Phép thay lui tương tự như phép thay tới. Cho U và y, ta giải phương trình thứ n trước rồi làm việc quay lui đến phương trình đầu tiên. Cũng như phép thay tới, tiến trình này chạy trong $\Theta(n^2)$ thời gian. Bởi U là tam giác trên, nên ta có thể viết lại hệ thống (31.22) là

$$\begin{array}{rcl} u_{11}x_1 + u_{12}x_2 + \dots + u_{1,n-2}x_{n-2} + u_{1,n-1}x_{n-1} + u_{1n}x_n & = & y_1 , \\ u_{22}x_2 + \dots + u_{2,n-2}x_{n-2} + u_{2,n-1}x_{n-1} + u_{2n}x_n & = & y_2 , \\ & \vdots & \\ u_{n-2,n-2}x_{n-2} + u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n & = & y_{n-2} , \\ u_{n-1,n-1}x_{n-1} + u_{n-1,n}x_n & = & y_{n-1} , \\ u_{n,n}x_n & = & y_n . \end{array}$$

Như vậy, ta có thể giải với $x_n, x_{n-1}, \ldots, x_1$ một cách thành công như sau:

$$x_n = y_n / u_{nn},$$

$$x_{n-1} = (y_{n-1} - u_{n-1,n}x_n) / u_{n-1,n-1},$$

$$x_{n-2} = (y_{n-2} - (u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n)) / u_{n-2,n-2},$$

hoặc, nói chung,

$$x_i = \left(y_i - \sum_{j=i+1}^{n} u_{ij}x_j \right) / u_{ii} .$$

Cho P, L, U, và b, thủ tục LUP-SOLVE giải với x bằng cách tổ hợp phép thay tới và lui. Mã giả mặc nhận chiều n xuất hiện trong thuộc tính *rows*[L] và rằng phép hoán vị ma trận P được biểu thị bởi mảng π.

LUP-SOLVE(L, U, π, b)

1 $n \leftarrow$ *rows*[L]

2 **for** $i \leftarrow 1$ **to** n

3 　　**do** $y_i \leftarrow b_{\pi[i]} - \sum_{j=1}^{i-1} l_{ij} y_j$

4 **for** $i \leftarrow n$ **downto** 1

5 　　**do** $x_i \leftarrow (y_i - \sum_{j=i+1}^{n} u_{ij}x_j) / u_{ii}$

6 **return** x

Thủ tục LUP-SOLVE giải với y dùng phép thay tới trong các dòng 2-3, rồi giải với x dùng phép thay lui trong các dòng 4-5. Bởi có một vòng lặp mặc định trong các phép lấy tổng trong từng vòng lặp **for**, nên thời gian thực hiện là $\Theta(n^2)$.

Để lấy ví dụ về các phương pháp này, ta xét hệ thống các phương trình tuyến tính được định nghĩa bởi

$$\begin{pmatrix} 1 & 2 & 0 \\ 3 & 5 & 4 \\ 5 & 6 & 3 \end{pmatrix} x = \begin{pmatrix} 0.1 \\ 12.5 \\ 10.3 \end{pmatrix},$$

ở đó

$$A = \begin{pmatrix} 1 & 2 & 0 \\ 3 & 5 & 4 \\ 5 & 6 & 3 \end{pmatrix}$$

$$b = \begin{pmatrix} 0.1 \\ 12.5 \\ 10.3 \end{pmatrix}$$

và ta muốn giải với x chưa biết. Phân tích LUP là

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 0.6 & 1 & 0 \\ 0.2 & 0.571 & 1 \end{pmatrix},$$

$$U = \begin{pmatrix} 5 & 6 & 3 \\ 0 & 1.4 & 2.2 \\ 0 & 0 & -1.856 \end{pmatrix}$$

$$P = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}.$$

(Bậcc giả có thể xác minh rằng $PA = LU$.) Dùng phép thay tới, ta giải $Ly = Pb$ với y:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0.6 & 1 & 0 \\ 0.2 & 0.571 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 10.3 \\ 12.5 \\ 0.1 \end{pmatrix},$$

có được

$$y = \begin{pmatrix} 10.3 \\ 6.32 \\ -5.569 \end{pmatrix}$$

bằng cách tính toán trước tiên là y_1, sau đó là y_2, và cuối cùng là y_3. Dùng phép thay lui, ta giải $Ux = y$ với x:

$$\begin{pmatrix} 5 & 6 & 3 \\ 0 & 1.4 & 2.2 \\ 0 & 0 & -1.856 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 10.3 \\ 6.32 \\ -5.569 \end{pmatrix}$$

và do đó có được đáp án mong muốn

$$x = \begin{pmatrix} 0.5 \\ -0.2 \\ 3.0 \end{pmatrix}$$

bằng cách tính x_3 trước, sau đó là x_2, và cuối cùng là x_1.

Tính toán một phân tích LU

Giờ đây sau khi ta chứng tỏ nếu có thể tính toán một phân tích LUP cho một ma trận phi lập dị A, phép thay tới và lui có thể được dùng để giải hệ thống $Ax = b$ của các phương trình tuyến tính. Ta chỉ cần nêu cách định cận hiệu quả một phân tích LUP cho A. Ta bắt đầu với trường hợp ở đó A là một ma trận phi lập dị $n \times n$ và P vắng mặt (hoặc, theo tương đương, $P = I_n$). Trong trường hợp này, ta phải tìm ra một phép thừa số hóa [factorization] $A = L\,U$. Ta gọi hai ma trận L và U là phân tích ***LU*** của A.

Tiến trình qua đó ta thực hiện phân tích LU được gọi là ***phép khử Gauss***. Để bắt đầu, ta trừ các bội của phương trình đầu tiên với các phương trình khác sao cho biến đầu tiên được gỡ bỏ ra khỏi các phương trình đó. Sau đó, ta trừ các bội của phương trình thứ hai với phương trình thứ ba và các phương trình khác sao cho giờ đây biến đầu tiên và thứ hai được gỡ bỏ ra khỏi chúng. Ta tiếp tục tiến trình này cho đến khi hệ thống còn lại sẽ có một dạng tam giác trên—thực tế, nó là ma trận U. Ma trận L được tạo dựng bởi các bộ nhân hàng khiến các biến được loại bỏ.

Thuật toán để thực thi chiến lược này là đệ quy. Ta muốn kiến tạo một phân tích LU cho một ma trận phi lập dị $n \times n$ A. Nếu $n = 1$, thì ta đã hoàn tất, bởi ta có thể chọn $L = I_1$, và $U = A$. Với $n > 1$, ta tách A thành bốn phần:

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

$$= \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ v & A' \end{pmatrix},$$

ở đó v là một vectơ cột kích cỡ $(n-1)$, w^{T} là một vectơ hàng kích cỡ $(n-1)$, và A' là một ma trận $(n-1) \times (n-1)$. Như vậy, dùng đại số học ma trận (xác minh các phương trình bằng cách đơn giản nhân qua), ta có thể lấy thừa số A là

$$A = \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ v & A' \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w_{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{11} \end{pmatrix}$$

Các 0 trong ma trận đầu tiên và thứ hai của phép thừa số hóa là các vectơ hàng và cột, theo thứ tự nêu trên, có kích cỡ $n-1$. Số hạng vw^{T}/a_{11}, *tất cả*, được lập thành bằng cách lấy tích ngoài của v và w rồi chia

từng thành phần của kết quả với a_{11}, là một ma trận $(n-1) \times (n-1)$, phù hợp theo kích cỡ với ma trận A' mà nó được trừ. Ma trận $(n-1) \times (n-1)$ kết quả

$$A' - vw^T/a_{11} \tag{31.23}$$

được gọi là ***phần bù Schur*** của A đối với a_{11}.

Giờ đây, ta tìm theo đệ quy một phân tích LU của phần bù Schur. Giả sử rằng

$$A' - vw^T/a_{11} = L'U',$$

ở đó L' là tam giác dưới đơn vị và U' là tam giác trên. Như vậy, dùng đại số học ma trận, ta có

$$\begin{aligned} A &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^T \\ 0 & A' - vw^T/a_{11} \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^T \\ 0 & L'U' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & L' \end{pmatrix} \begin{pmatrix} a_{11} & w^T \\ 0 & U' \end{pmatrix} \\ &= LU, \end{aligned}$$

và do đó có phân tích LU của chúng ta. (Lưu ý, bởi L' là tam giác dưới đơn vị, nên L cũng vậy, và bởi U' là tam giác trên, nên U cũng vậy.)

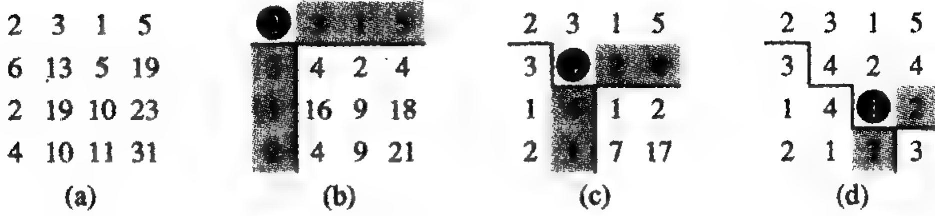
Tất nhiên, nếu $a_{11} = 0$, phương pháp này không làm việc, bởi nó chia cho 0. Nó cũng không làm việc nếu khoản nhập mút trên trái của phần bù Schur $A' - vw^T/a_{11}$ là 0, bởi ta chia cho nó trong bước kế tiếp của đệ quy. Các thành phần mà ta chia với trong khi chạy phân tích LU được gọi là các ***trục quay*** [pivot], và chúng choán các thành phần đường chéo của ma trận U. Lý do mà ta gộp một ma trận hoán vị P trong khi chạy phân tích LUP đó là nó cho phép ta tránh chia với các thành phần zero. Việc dùng các phép hoán vị để tránh phép chia 0 (hoặc chia với các số nhỏ) được gọi là ***phép quay trục*** [pivoting].

Một lớp quan trọng các ma trận mà phân tích LU luôn làm việc đúng đắn đó là lớp các ma trận xác định dương, đối xứng. Các ma trận như vậy không yêu cầu phép quay trục, và như vậy có thể sử dụng chiến lược đệ quy nêu trên đây mà không sợ chia 0. Ta sẽ chứng minh kết quả này, cũng như vài kết quả khác, trong Đoạn 31.6.

Mã của phân tích LU của một ma trận A tuân thủ chiến lược đệ quy, ngoại trừ một lần lặp lại vòng lặp sẽ thay bước đệ quy đó. (Phép biến đổi này là một kiểu tối ưu hóa chuẩn cho một thủ tục "đệ quy đuôi"—mà phép toán của nó là một lệnh gọi đệ quy lên chính nó.) Mặc nhận rằng chiều của A được lưu giữ trong thuộc tính *rows*[A]. Bởi ta biết rằng kết xuất ma trận U có các 0 bên dưới đường chéo, và bởi LU-SOLVE không xem xét các khoản nhập này, nên mã không quan tâm đến việc điền chúng. Cũng vậy, bởi ma trận kết xuất L có các 1 đường chéo củanó và các 0 bên trên đường chéo, các khoản nhập này cũng không được điền. Như vậy, mã chỉ tính toán các khoản nhập "quan trọng" của L và U.

```
LU-DECOMPOSITION(A)
1   n ← rows[A]
2   for k ← 1 to n
3        do u_kk ← a_kk
4           for i ← k+1 to n
5               do l_ik ← a_ik/u_kk          ▷ l_ik lưu giữ v_i
6                  u_ki ← a_ki               ▷ u_ki lưu giữ w^T
7           for i ← k+1 to n
8               do for j ← k + 1 to n
9                      do a_ij ← a_ij − l_ik u_kj
10  return L và U
```

Vòng lặp **for** ngoài bắt đầu tại dòng 2 sẽ lặp lại một lần cho từng bước đệ quy. Bên trong vòng lặp này, trục quay được xác định là $u_{kk} = a_{kk}$ tại dòng 3. Bên trong vòng lặp **for** tại các dòng 4-6 (không thi hành khi $k = n$), các vectơ v và w^T được dùng để cập nhật L và U. Các thành phần của vectơ v được xác định trong dòng 5, ở đó v_i được lưu trữ trong l_{ik}, và các thành phần của vectơ w^T được xác định trong dòng 6, ở đó w^T được lưu trữ trong u_{ki}. Cuối cùng, các thành phần của phần bù Schur được tính toán trong các dòng 7-9 và được lưu trữ trở lại trong ma trận A. Bởi dòng 9 được lồng ba lần, LU-DECOMPOSITION chạy trong thời gian $\Theta(n^3)$.



$$\underbrace{\begin{pmatrix} 2 & 3 & 1 & 5 \\ 6 & 13 & 5 & 19 \\ 2 & 19 & 10 & 23 \\ 4 & 10 & 11 & 31 \end{pmatrix}}_{A} = \underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 \\ 1 & 4 & 1 & 0 \\ 2 & 1 & 7 & 1 \end{pmatrix}}_{L} \underbrace{\begin{pmatrix} 2 & 3 & 1 & 5 \\ 0 & 4 & 2 & 4 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 3 \end{pmatrix}}_{U}$$

Hình 31.1 Phép toán của LU-DECOMPOSITION. **(a)** Ma trận A. **(b)** Thành phần $a_{11} = 2$ màu đen là trục quay, cột tô bóng là v/a_{11}, và hàng tô bóng là w^T. Các thành phần của U được tính toán cho đến nay đều nằm bên trên vạch ngang, và các thành phần của L nằm bên trái của vạch dọc. Ma trận phần bù Schur $A' - vw^T/a_{11}$ choán phần dưới phải. **(c)** Giờ đây ta hoạt động trên ma trận phần bù Schur được tạo từ phần (b). Thành phần $a_{22} = 4$ tô đen là trục quay, và cột và hàng tô bóng là v/a_{22} và w^T (trong tiến trình phân hoạch của phần bù Schur), theo thứ tự nêu trên. Các vạch chia ma trận thành các thành phần của U được tính toán cho đến giờ (bên trên), các thành phần của L đã tính toán cho đến giờ (bên trái), và phần bù Schur mới (dưới phải). **(d)** Bước kế tiếp hoàn tất phép thừa số hóa. (Thành phần 3 trong phần bù Schur mới trở thành phần của U khi đệ quy kết thúc.) **(e)** Phép thừa số hóa $A = LU$.

Hình 31.1 minh họa phép toán của LU-DECOMPOSITION. Nó nêu một kiểu tối ưu hóa chuẩn của thủ tục ở đó các thành phần quan trọng của L và U được lưu trữ "tại chỗ" trong ma trận A. Nghĩa là, ta có thể xác lập một tương ứng giữa mỗi thành phần a_{ij} và l_{ij} (nếu $i > j$) hoặc u_{ij} (nếu $i \leq j$) rồi cập nhật ma trận A sao cho nó lưu giữ cả L lẫn U khi thủ tục kết thúc. Mã giả cho phép tối ưu hóa này có được từ mã giả trên đây bằng cách đơn thuần thay mỗi tham chiếu đến l hoặc u bằng a; chẳng có gì khó khăn khi xác minh phép biến đổi này bảo toàn tính đúng đắn.

Tính toán một phân tích LUP

Nói chung, trong khi giải một hệ thống phương trình tuyến tính $Ax = b$, trên các thành phần ngoài đường chéo của A để tránh chia 0. Không những là phép chia 0 không thỏa đáng, mà cả phép chia với một giá trị nhỏ bất kỳ, cho dù A là phi lập dị, bởi các tính không ổn định về số có thể dẫn đến phép tính. Do đó ta gắng quay trục [pivot] trên một giá trị lớn.

Toán học đàng sau phân tích LUP cũng tương tự như toán học của

phân tích LU. Hãy nhớ lại ta có một ma trận phi lập dị $n \times n$ A và muốn tìm ra một ma trận hoán vị P, một ma trận tam giác dưới đơn vị L, và một ma trận tam giác trên U sao cho $PA = L\ U$. Trước khi tả phân hoạch ma trận A, như đã thực hiện với phân tích LU, ta dời một thành phần phi zero, giả sử a_{k1}, từ cột đầu tiên đến vị trí (1, 1) của ma trận. (Nếu cột đầu tiên chỉ chứa các 0, thì A là lập dị, bởi định thức củanó là 0, theo các Định lý 31.4 và 31.5.) Để bảo toàn tập hợp các phương trình, ta trao đổi hàng 1 with hàng k, tương đương với việc nhân A với một ma trận hoán vị Q bên trái (Bài tập 31.1-2). Như vậy, ta có thể viết QA dưới dạng

$$QA = \begin{pmatrix} a_{k1} & w^T \\ v & A' \end{pmatrix},$$

ở đó $v = (a_{21}, a_{31},..., a_{n1})^T$, ngoại trừ a_{11}, thay a_{k1}; $w^T = (a_{k2}, a_{k3},..., a_{kn})$; và A' là một ma trận $(n-1) \times (n-1)$. Bởi $a_{k1} \neq 0$, giờ đây ta có thể thực hiện cùng đại số học tuyến tính tương tự như trường hợp phân tích LU, nhưng giờ đây bảo đảm ta không chia 0:

$$\begin{aligned} QA &= \begin{pmatrix} a_{k1} & w^T \\ v & A' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^T \\ 0 & A' - vw^T/a_{k1} \end{pmatrix}. \end{aligned}$$

Phần bù Schur $A' - vw^T/a_{k1}$ là phi lập dị, bởi bằng không ma trận thứ hai trong phương trình cuối có định thức 0, và như vậy định thức của ma trận A là 0; nhưng điều này có nghĩa A là lập dị, mâu thuẫn với giả thiết của chúng ta rằng A là phi lập dị. Bởi vậy, theo quy nạp ta có thể tìm một phân tích LUP cho phần bù Schur, với ma trận tam giác dưới đơn vị L', ma trận tam giác trên U', và ma trận hoán vị P', sao cho

$$P'(A' - vw^T/a_{k1}) = L'U'.$$

Định nghĩa

$$P = \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} Q,$$

là một ma trận hoán vị, bởi nó là tích của hai ma trận hoán vị (Bài tập 31.1-2). Giờ đây, ta có

$$\begin{aligned} PA &= \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} QA \\ &= \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} \begin{pmatrix} 1 & 0 \\ v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w_1 \\ 0 & A' - vw^T/a_{k1} \end{pmatrix} \end{aligned}$$

$$
\begin{aligned}
&= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & P' \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{k1} \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & P'(A' - vw^{\mathrm{T}}/a_{k1}) \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & L'U' \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & L' \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & U' \end{pmatrix} \\
&= LU ,
\end{aligned}
$$

cho ra phân tích LUP. Bởi L' là đơn vị tam giác dưới, do đó là L, và bởi U' là tam giác trên, do đó là U.

Lưu ý, trong phép lấy đạo hàm này, khác với trường hợp của phân tích LU, cả hai vectơ cột v/a_{k1} và phần bù Schur $A' - vw^{\mathrm{T}}/a_{k1}$ phải được nhân với ma trận hoán vị P'.

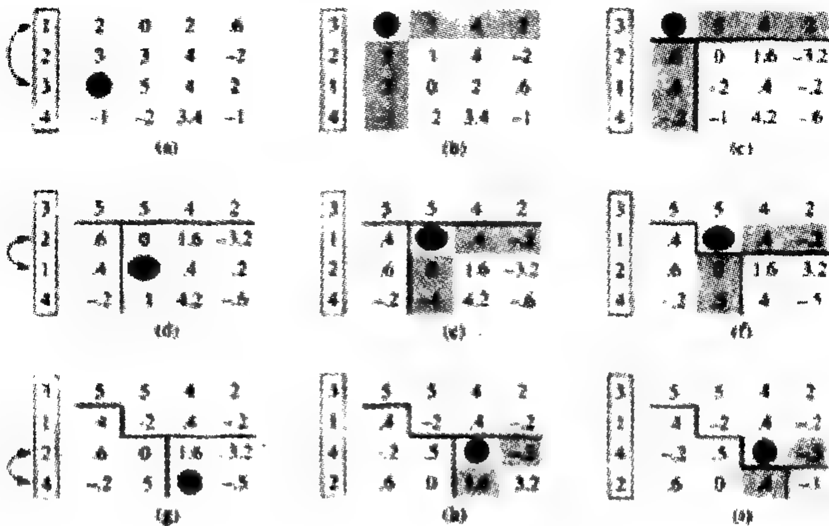
Cũng như LU-DECOMPOSITION, mã giả của chúng ta để phân tích LUP sẽ thay đệ quy bằng một lần lặp lại vòng lặp. Như là một cải tiến so với một thực thi trực tiếp của đệ quy, ta năng bậcng duy trì ma trận hoán vị P dưới dạng một mảng n, ở đó $\pi[i] = j$ có nghĩa là hàng thứ i của P chứa một 1 trong cột j. Ta cũng thực thi mã để tính toán L và U "tại chỗ" trong ma trận A. Như vậy, khi thủ tục kết thúc,

$$
a_{ij} = \begin{cases} l_{ij} & \text{nếu } i > j, \\ u_{ij} & \text{nếu } i \le j . \end{cases}
$$

```
LUP-DECOMPOSITION(A)
n ← rows[A]
for i ← 1 to n
    do π[i] ← i
for k ← 1 to n − 1
    do p ← 0
       for i ← k to n
           do if |a_ik| > p
                 then p ← |a_ik|
                      k' ← i
       if p = 0
           then error "singular matrix"
       trao đổi π[k] ← π[k']
```

```
13        for i ← 1 to n
14            do trao đổi a_ki ↔ a_k'i
15        for i ← k + 1 to n
16            do a_ik ← a_ik / a_kk
17                for j ← k + 1 to n
18                    do a_ij ← a_ij − a_ik a_kj
```

Hình 31.2 minh họa cách thức mà LUP-DECOMPOSITION lấy thừa số một ma trận. Mảng π được khởi tạo bởi các dòng 2-3 để biểu diễn phép hoán vị đồng nhất thức. Vòng lặp **for** ngoài bắt đầu tại dòng 4 thực thi độ quy. Mỗi lần qua vòng lặp ngoài, các dòng 5-9 xác định thành phần $a_{k'k}$ với giá trị tuyệt đối lớn nhất của những giá trị trong cột đầu tiên hiện hành (cột k) của ma trận $(n - k+ 1) \times (n - k+ 1)$ mà phân tích LU của nó phải được tìm thấy.



Hình 31.2 Phép toán của LUP-DECOMPOSITION. **(a)** Ma trận đầu vào A với phép hoán vị đồng nhất thức của các hàng bên trái. Bước đầu tiên của thuật toán xác định thành phần 5 màu đen trong hàng thứ ba là trục quay cho cột đầu tiên. **(b)** Các hàng 1 và 3 được tráo và phép hoán vị được cập nhật. Cột và hàng tô bóng biểu diễn v và w^T. **(c)** Vectơ v được thay bằng $v/5$, và dưới phải của ma trận được cập nhật theo phần bù Schur. Các dòng chia ma trận thành ba vùng: các thành phần của U (bên trên), các thành phần của L (trái), và các thành phần của phần bù Schur (dưới phải). **(d)-(f)** Bước thứ hai. **(g)-(i)** Bước thứ ba hoàn tất thuật toán. *(j)* Phân tích LUP $PA = LU$.

Nếu tất cả các thành phần trong cột đầu tiên hiện hành là zero, các dòng 10-11 báo cáo ma trận là lập dị. Để quay trục, ta đổi $\pi[k']$ với $\pi[k]$ trong dòng 12 và đổi các hàng thứ k và k' của A trong các dòng 13-14, và do đó tạo thành phần trục $a_{kk'}$. (Toàn bộ các hàng được tráo bởi vì theo phép lấy đạo hàm của phương pháp trên đây, không những là $A' - vw^T/a_{k1}$ được nhân với P', mà còn là v/a_{k1}.) Cuối cùng, phần bù Schur được tính toán bởi các dòng 15-18 tương tự như được tính toán bởi các dòng 4-9 của LU-DECOMPOSITION, ngoại trừ ở đây phép toán được viết để làm việc "tại chỗ."

Do cấu trúc vòng lặp được lồng ba lần củanó, thời gian thực hiện của LUP-DECOMPOSITION là $\Theta(n^3)$, giống như của LU-DECOMPOSITION. Như vậy, mức hao phí của phép quay trục tối đa bằng một thừa số bất biến về thời gian.

Bài tập

31.4-1

Giải phương trình

$$\begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ -6 & 5 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ 14 \\ -7 \end{pmatrix}$$

dùng phép thay tới.

31.4-2

Tìm một phân tích LU của ma trận

$$\begin{pmatrix} 4 & -5 & 6 \\ 8 & -6 & 7 \\ 12 & -7 & 12 \end{pmatrix} .$$

31.4-3

Tại sao vòng lặp **for** trong dòng 4 của LUP-DECOMPOSITION chỉ chạy tới $n - 1$, trong khi đó vòng lặp **for** tương ứng trong dòng 2 của LU-DECOMPOSITION chạy hết tới n?

31.4-4

Giải phương trình

$$\begin{pmatrix} 1 & 5 & 4 \\ 2 & 0 & 3 \\ 5 & 8 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 12 \\ 9 \\ 5 \end{pmatrix}$$

dùng một phân tích LUP.

31.4-5

Mô tả phân tích LUP của một ma trận đường chéo.

31.4-6

Mô tả phân tích LUP của một ma trận hoán vị A, và chứng minh nó là duy nhất.

31.4-7

Chứng tỏ với tất cả $n \geq 1$, ở đó tồn tại các ma trận $n \times n$ lập dị có các phân tích LU.

31.4-8 *

Nêu cách giải hiệu quả một tập hợp các phương trình có dạng $Ax = b$ trên tựa vành bool $(\{0, 1\}, \vee, \wedge, 0,1)$.

31.4-9 *

Giả sử A là một ma trận thực $m \times n$ có hạng m, ở đó $m < n$. Nêu cách tìm vectơ x_0 kích cỡ n và một ma trận $m \times (n - m)$ B có hạng $n - m$ sao cho mọi vectơ có dnạg $x_0 + By$, với $y \in \mathbf{R}^{n-m}$, là một giải pháp cho phương trình xác định dưới $Ax = b$.

31.5 Đảo các ma trận

Mặc dù trong thực tế ta thường không dùng các phép đảo ma trận để giải các hệ thống phương trình tuyến tính, thay vì thế người ta ưa dùng các kỹ thuật ổn định về số hơn như phân tích LUP, song đôi lúc ta cũng cần phải tính toán một ma trận nghịch đảo. Trong đoạn này, ta nêu cách dùng phân tích LUP để tính toán một ma trận nghịch đảo. Ta cũng đề cập vấn đề đáng quan tâm về lý thuyết đó là liệu phép tính của một ma trận nghịch đảo có thể tăng tốc bằng các kỹ thuật như thuật toán Strassen cho phép nhân ma trận. Quả vậy, tài liệu gốc của Strassen đã được thúc đẩy bởi bài toán chứng tỏ một tập hợp các phương trình tuyến tính có thể được giải nhanh hơn so với phương pháp bình thường.

Tính toán một ma trận nghịch đảo từ một phân tích LUP

Giả sử ta có một phân tích LUP của một ma trận A theo dạng ba ma trận L, U, và P sao cho $PA = LU$. Dùng LU-SOLVE, ta có thể giải một phương trình có dạng $Ax = b$ trong thời gian $\Theta(n^2)$. Bởi phân tích LUP tùy thuộc vào A chứ không phải b, nên ta có thể giải một tập hợp thứ hai

các phương trình có dạng $Ax = b'$ trong thời gian bổ sung $\Theta(n^2)$. Nói chung, sau khi có phân tích LUP của A, ta có thể giải, trong thời gian $\Theta(kn^2)$, k phiên bản của phương trình $Ax = b$ mà chỉ khác trong b.

Phương trình

$$AX = I_n \tag{31.24}$$

có thể được xem là một tập hợp n phương trình riêng biệt có dạng $Ax = b$. Các phương trình này định nghĩa ma trận X dưới dạng nghịch đảo của A. Để chính xác, cho X_i thể hiện cột thứ i của X, và hãy nhớ lại vectơ đơn vị e_i là cột thứ i của I_n. Như vậy, phương trình (31.24) có thể được giải với X bằng cách dùng phân tích LUP của A để giải từng phương trình

$$AX_i = e_i$$

một cách tách biệt với X_i. Mỗi trong số n X_i có thể tìm thấy trong thời gian $\Theta(n^2)$, và do đó phép tính của X từ phân tích LUP của A mất một thời gian $\Theta(n^3)$. Bởi phân tích LUP của A có thể được tính toán trong thời gian $\Theta(n^3)$, nghịch đảo A^{-1} của một ma trận A có thể được xác định trong thời gian $\Theta(n^3)$.

Phép nhân ma trận và phép nghịch đảo ma trận

Giờ đây, ta chứng tỏ các tăng tốc lý thuyết có được cho phép nhân ma trận phiên dịch thành các tăng tốc cho phép nghịch đảo ma trận. Thực vậy, ta chứng minh một cái gì đó mạnh hơn: phép nghịch đảo ma trận tương đương với phép nhân ma trận, theo nghĩa sau. Nếu $M(n)$ thể hiện thời gian để nhân hai ma trận $n \times n$ và $I(n)$ thể hiện thời gian để đảo một ma trận phi lập dị $n \times n$, thì $I(n) = \Theta(M(n))$. Ta chứng minh điều này dẫn đến hai phần. Thứ nhất, ta chứng tỏ $M(n) = O(I(n))$, tương đối dễ dàng, rồi ta chứng minh $I(n) = O(M(n))$.

Định lý 31.11 (Phép nhân không khó hơn phép nghịch đảo)

Nếu ta có thể đảo một ma trận $n \times n$ trong thời gian $I(n)$, ở đó $I(n) = \Omega(n^2)$ và $I(n)$ thỏa điều kiện tính đều $I(3n) = O(I(n))$, thì ta có thể nhân hai ma trận $n \times n$ trong thời gian $O(I(n))$.

Chứng minh Cho A và B là các ma trận $n \times n$ có tích ma trận C mà ta muốn tính toán. Ta định nghĩa ma trận $3n \times 3n$ D bằng

$$D = \begin{pmatrix} I_n & A & 0 \\ 0 & I_n & B \\ 0 & 0 & I_n \end{pmatrix}.$$

Nghịch đảo của D là

$$D^{-1} = \begin{pmatrix} I_n & -A & AB \\ 0 & I_n & -B \\ 0 & 0 & I_n \end{pmatrix},$$

và như vậy ta có thể tính toán tích AB bằng cách ma trận con $n \times n$ trên phải của D^{-1}.

Ta có thể kiến tạo ma trận D trong $\Theta(n^2) = O(I(n))$ thời gian, và ta có thể đảo D trong $O(I(3n)) = O(I(n))$ thời gian, theo điều kiện tính đều trên $I(n)$. Như vậy ta có

$M(n) = O(I(n))$.

Lưu ý, $I(n)$ thỏa điều kiện tính đều mỗi khi $I(n)$ không có các lần nhảy lớn về giá trị. Ví dụ, nếu $I(n) = \Theta(n^c \lg^d n)$ với bất kỳ các hằng $c > 0$, $d \geq 0$, thì $I(n)$ thỏa điều kiện tính đều.

Rút gọn phép nghịch đảo ma trận thành phép nhân ma trận

Phần chứng minh rằng phép nghịch đảo ma trận không khó hơn phép nhân ma trận dựa vào vài tính chất của các ma trận xác định-dương đối xứng mà ta sẽ chứng minh trong Đoạn 31.6.

Định lý 31.12 (Phép đảo không khó hơn phép nhân)

Nếu có thể nhân hai ma trận thực $n \times n$ trong thời gian $M(n)$, ở đó $M(n) = \Omega(n^2)$ và $M(n) = O(M(n + k))$ với $0 \leq k \leq n$, thì ta có thể tính toán nghịch đảo của bất kỳ ma trận thực phi lập dị $n \times n$ nào trong thời gian $O(M(n))$.

Chứng minh Ta có thể mặc nhận rằng n là một lũy thừa chính xác của 2, bởi ta có

$$\begin{pmatrix} A & 0 \\ 0 & I_k \end{pmatrix}^{-1} = \begin{pmatrix} A^{-1} & 0 \\ 0 & I_k \end{pmatrix}$$

với bất kỳ $k > 0$. Như vậy, bằng cách chọn k sao cho $n + k$ là một lũy thừa của 2, ta mở rộng ma trận đến một kích cỡ là lũy thừa kế tiếp của 2 và được đáp án mong muốn A^{-1} từ đáp án của bài toán đã mở rộng. Điều kiện tính đều trên $M(n)$ bảo đảm sự mở rộng này không làm cho thời gian thực hiện tăng lên theo nhiều thừa số bất biến.

Trước mắt, ta hãy mặc nhận ma trận $n \times n$ A là đối xứng và xác định dương. Ta phân hoạch A thành bốn ma trận con $n/2 \times n/2$:

$$A = \begin{pmatrix} B & C^T \\ C & D \end{pmatrix} \qquad (31.25)$$

Như vậy, nếu ta cho

$$S = D - CB^{-1}C^{T} \qquad (31.26)$$

là phần bù Schur của A đối với B, ta có

$$A^{-1} = \begin{pmatrix} B^{-1} + B^{-1}C^{T}S^{-1}CB^{-1} & -B^{-1}C^{T}S^{-1} \\ -S^{-1}CB^{-1} & S^{-1} \end{pmatrix}, \qquad (31.27)$$

bởi $AA^{-1} = I_n$, như có thể xác minh bằng cách thực hiện phép nhân ma trận. Các ma trận B^{-1} và S^{-1} tồn tại nếu A là đối xứng và xác định dương, theo các Bổ đề 31.13, 31.14, và 31.15 trong Đoạn 31.6, bởi cả B lẫn S là đối xứng và xác định dương. Theo Bài tập 31.1-3, $B^{-1}C^{T} = (CB^{-1})^{T}$ và $B^{-1}C^{T}S^{-1} = (S^{-1}CB^{-1})^{T}$. Do đó, các phương trình (31.26) và (31.27) có thể được dùng để đặc tả một thuật toán đệ quy bao hàm 4 phép nhân của các ma trận $n/2 \times n/2$:

$C \bullet B^{-1}$,

$(CB^{-1}) \bullet C^{T}$,

$S^{-1} \bullet (CB^{-1})$,

$(CB^{-1})^{T} \bullet (S^{-1}CB^{-1})$.

Bởi ta có thể nhân các ma trận $n/2 \times n/2$ dùng một thuật toán cho các ma trận $n \times n$, phép nghịch đảo ma trận của các ma trận xác định dương đối xứng có thể được thực hiện trong thời gian

$$\begin{aligned} I(n) &\le 2I(n/2) + 4M(n) + O(n^2) \\ &= 2I(n/2) + O(M(n)) \\ &= O(M(n)) . \end{aligned}$$

Còn lại ta chỉ cần chứng minh thời gian thực hiện tiệm cận của phép nhân ma trận có thể có được đối với phép nghịch đảo ma trận khi A là đảo nghịch nhưng không đối xứng và xác định dương. Ý tưởng căn bản đó là với bất kỳ ma trận phi lập dị A, ma trận $A^{T}A$ là đối xứng (theo Bài tập 31.1-3) và xác định dương (theo Định lý 31.6). như vậy, điểm tinh tế đó là rút gọn bài toán đảo A thành bài toán đảo $A^{T}A$.

Phép rút gọn dựa trên nhận xét rằng khi A là một ma trận phi lập dị $n \times n$, ta có

$$A^{-1} = (A^{T}A)^{-1}A^{T}$$

bởi $((A^{T}A)^{-1}A^{T})A = (A^{T}A)^{-1}(A^{T}A) = I_n$ và một ma trận nghịch đảo là duy nhất. Do đó, ta có thể tính toán A^{-1} bằng cách trước tiên nhân A^{T} với A để được $A^{T}A$, sau đó đảo ma trận xác định dương đối xứng $A^{T}A$ dùng thuật toán chia để trị trên đây, và cuối cùng nhân kết quả với A^{T}. Mỗi trong ba bước này chiếm $O(M(n))$ thời gian, và như vậy bất kỳ ma trận phi lập dị với các khoản nhập thực có thể được đảo trong $O(M(n))$ thời gian.

Phần chứng minh của Định lý 31.12 gợi ý một biện pháp để giải phương trình $Ax = b$ mà không dùng phép quay trục, miễn là A là phi lập dị. Ta nhân cả hai cạnh của phương trình với A^T, cho ra $(A^TA)x = A^Tb$. Phép biến đổi này không ảnh hưởng giải pháp x, bởi A^T là đảo nghịch, do đó ta có thể lấy thừa số ma trận xác định dương đối xứng $A^T A$ bằng cách tính toán một phân tích LU. Sau đó, ta dùng phép thay tới và lui để giải với x với phía bên phải $A^T b$. Mặc dù phương pháp này đúng về lý thuyết, song trong thực tế thủ tục LUP-DECOMPOSITION làm việc tốt hơn nhiều. Phân tích LUP yêu cầu ít phép toán số học hơn theo một thừa số bất biến, và nó có các tính chất số hơi tốt hơn.

Bài tập

31.5-1

Cho $M(n)$ là thời gian để nhân các ma trận $n \times n$, và cho $S(n)$ thể hiện thời gian cần thiết để bình phương một ma trận $n \times n$. Chứng tỏ việc nhân và bình phương các ma trận chủ yếu cũng khó như nhau: $S(n) = \Theta(M(n))$.

31.5-2

Cho $M(n)$ là thời gian để nhân các ma trận $n \times n$, và cho $L(n)$ là thời gian để tính toán phân tích LUP của một ma trận $n \times n$. Chứng tỏ việc nhân các ma trận và tính toán các phân tích LUP của các ma trận cũng khó như nhau: $L(n) = \Theta(M(n))$.

31.5-3

Cho $M(n)$ là thời gian để nhân các ma trận $n \times n$, và cho $D(n)$ thể hiện thời gian cần thiết để tìm ra định thức của một ma trận $n \times n$. Chứng tỏ việc tìm định thức không khó hơn việc nhân các ma trận: $D(n) = O(M(n))$.

31.5-4

Cho $M(n)$ là thời gian để nhân các ma trận bool $n \times n$, và cho $T(n)$ là thời gian để tìm ra bao đóng bắc cầu của các ma trận bool $n \times n$. Chứng tỏ $M(n) = O(T(n))$ và $T(n) = O(M(n) \lg n)$.

31.5-5

Thuật toán ma trận-nghịch đảo dựa trên Định lý 31.12 có làm việc không khi các thành phần ma trận được rút từ trường các số nguyên modulo 2? Giải thích.

31.5-6 *

Tổng quát hóa thuật toán ma trận-nghịch đảo của Định lý 31.12 để

điều quản các ma trận của các số phức, và chứng minh phép tổng quát hóa của bạn làm việc đúng đắn. (*Mách nước:* Thay vì phép chuyển vị của A, bạn dùng phép ***chuyển vị liên hợp*** A^*, có được từ phép chuyển vị của A bằng cách thay mọi khoản nhập bằng liên hợp phức của nó. Thay vì các ma trận đối xứng, hãy xét các ma trận ***Hermitian***, là các ma trận A sao cho $A = A^*$.)

31.6 Các ma trận xác định dương đối xứng và phép xấp xỉ các bình phương bé nhất

Các ma trận xác định dương đối xứng có nhiều tính chất mong muốn và đáng quan tâm. Ví dụ, chúng là phi lập dị, và phân tích LU có thể được thực hiện trên chúng mà ta không phải lo nghĩ về phép chia 0. Trong đoạn này, ta sẽ chứng minh vài tính chất quan trọng khác của các ma trận xác định dương đối xứng và nêu một ứng dụng đáng quan tâm về cách vẽ đường cong theo các điểm bằng phép xấp xỉ các bình phương bé nhất.

Tính chất đầu tiên mà ta chứng minh có lẽ là cơ bản nhất.

Bổ đề 31.13

Mọi ma trận xác định dương đối xứng đều là phi lập dị.

Chứng minh Giả sử một ma trận A là lập dị. Thì theo Hệ luận 31.3, sẽ tồn tại một vectơ phi zero x sao cho $Ax = 0$. Do đó, $x^T Ax = 0$, và A không thể là xác định dương.

Ta sẽ chú ý đến phần chứng minh rằng ta có thể thực hiện phân tích LU trên một ma trận xác định dương đối xứng A mà không chia 0. Để bắt đầu, ta chứng minh các tính chất về một số ma trận con của A. Định nghĩa ***ma trận con dẫn đầu*** [leading submatrix] thứ k của A là ma trận A_k bao gồm phần giao của các hàng k đầu tiên và các cột k đầu tiên của A.

Bổ đề 31.14

Nếu A là một ma trận xác định dương đối xứng, thì mọi ma trận con dẫn đầu của A là đối xứng và xác định dương.

Chứng minh Rằng mỗi ma trận con dẫn đầu A_k là đối xứng là hiển nhiên. Để chứng minh rằng A_k là xác định dương, ta cho x là một vectơ cột phi zero có kích cỡ k, và ta hãy phân hoạch A như sau:

$$A = \begin{pmatrix} A_k & B^T \\ B & C \end{pmatrix}.$$

Thì, ta có

$$x^{T}A_k x = (x^T\ 0)\begin{pmatrix} A_k & B^T \\ B & C \end{pmatrix}\begin{pmatrix} x \\ 0 \end{pmatrix}$$

$$= (x^T\ \ 0)A\begin{pmatrix} x \\ 0 \end{pmatrix}$$

$$> 0,$$

bởi A là xác định dương, và do đó A_k cũng là xác định dương.

Giờ đây, ta chuyển sang vài tính chất chủ yếu của phần bù Schur. Cho A là một ma trận xác định dương đối xứng, và cho A_k là một ma trận con dẫn đầu $k \times k$ của A. Phân hoạch A dưới dạng

$$A = \begin{pmatrix} A_k & B^T \\ B & C \end{pmatrix}. \qquad (31.28)$$

Như vậy, ***phần bù Schur*** của A đối với A_k được định nghĩa là

$$S = C - BA_k^{1} B^T. \qquad (31.29)$$

(Theo Bổ đề 31.14, A_k là đối xứng và xác định dương; do đó, A_k^{-1} tồn tại theo Bổ đề 31.13, và S được định nghĩa kỹ.) Lưu ý, phần định nghĩa trên đây của chúng ta (31.23) về phần bù Schur là nhất quán với (31.29), bằng cách cho $k = 1$.

Bổ đề kế tiếp chứng tỏ bản thân các ma trận phần bù Schur của các ma trận xác định dương đối xứng cũng đối xứng và xác định dương. Kết quả này đã được dùng trong Định lý 31.12, và hệ luận củanó cần chứng minh tính đúng đắn của phân tích LU đối với các ma trận xác định dương đối xứng.

Bổ đề 31.15 (Bổ đề phần bù Schur)

Nếu A là một ma trận xác định dương đối xứng và A_k là một ma trận con dẫn đầu $k \times k$ của A, thì phần bù Schur của A đối với A_k là đối xứng và xác định dương.

Chứng minh Rằng S là đối xứng là của Bài tập 31.1-7. Ta chỉ còn phải chứng tỏ S là xác định dương. Xét phân hoạch của A đã cho trong phương trình (31.28).

Với bất kỳ vectơ phi zero x, ta có $x^T Ax > 0$ theo giả thiết. Ta hãy tách x thành hai vectơ con y và z tương thích với A_k và C, theo thứ tự nêu trên. Bởi A_k^{-1} tồn tại, nên ta có

$$x^T Ax = (y^T z^T)\begin{pmatrix} A_k & B^T \\ B & C \end{pmatrix}\begin{pmatrix} y \\ z \end{pmatrix}$$

$$= y^T A_k y + y^T B^T z + z^T By + z^1 Cz$$

$$= (y + A_k^{-1}B^{T}z)^{T}A_k(y + A_k^{-1}B^{T}z) + z^{T}(C - BA_k^{-1}B^{T})z, \qquad (31.30)$$

theo ma thuật ma trận. (Hãy xác minh bằng cách nhân qua.) Phương trình cuối này chung quy là "hoàn tất bình phương" theo dạng bậc hai. (Xem Bài tập 31.6-2.)

Bởi $x^T Ax > 0$ áp dụng cho mọi x phi zero, nên ta hãy chọn một z phi zero rồi chọn $y = -A_k^{-1} B^T z$, điều này khiến số hạng đầu tiên trong phương trình (31.30) triệt tiêu, để lại

$$z^T (C - BA_k^{-1} B^T)z = z^T Sz$$

làm giá trị của biểu thức. Do đó, với bất kỳ $z \neq 0$, ta có $z^T Sz = x^T Ax > 0$, và như vậy S là xác định dương.

Hệ luận 31.16

Phân tích LU của một ma trận xác định dương đối xứng không bao giờ gây ra một phép chia 0.

Chứng minh Cho A là một ma trận xác định dương đối xứng. Ta sẽ chứng minh một cái gì đó mạnh hơn phát biểu của hệ luận: mọi trục quay là dương ngặt. Trục quay đầu tiên là a_{11}. Cho e_1 là vectơ đơn vị đầu tiên, từ đó ta được $a_{11} = e^T Ae_1 > 0$. Bởi bước đầu tiên của phân tích LU sẽ tạo ra phần bù Schur của A đối với $A_1 = (a_{11})$, Bổ đề 31.15 hàm ý rằng tất cả các trục quay là dương theo phương pháp quy nạp.

Phép xấp xỉ các bình phương bé nhất

Việc vẽ các đường cong theo các điểm theo các tập hợp các điểm dữ liệu đã cho là một ứng dụng quan trọng của các ma trận xác định dương đối xứng. Giả sử ta có một tập hợp m điểm dữ liệu

$$(x_1, y_1),\ (x_2, y_2), ..., (x_m, y_m)\ ,$$

ở đó y_i được xem là phải tuân thủ các lỗi đo. Ta muốn xác định một hàm $F(x)$ sao cho

$$y_i = F(x_i) + \eta_i\ , \qquad (31.31)$$

với $i = 1, 2,..., m$, ở đó các lỗi xấp xỉ η_i là nhỏ. Dạng của hàm F tùy thuộc vào bài toán đang có. Ở đây, ta mặc nhận nó có dạng của một tổng gia trọng theo tuyến tính,

$$F(x) = \sum_{j=1}^{n} c_j f_j(x)\ ,$$

ở đó số lượng các số hạng n và các ***hàm cơ bản*** f_j cụ thể được chọn dựa trên kiến thức của bài toán đang có. Một chọn lựa chung đó là $f_j(x) = x^{j-1}$, có nghĩa là

$$F(x) = c_1 + c_2x + c_3x^2 + ... + c_nx^{n-1}$$

là một đa thức có bậc $n - 1$ trong x.

Bằng cách chọn $n = m$, ta có thể tính toán mỗi y_i *một cách chính xác* trong phương trình (31.31). Tuy nhiên, một F bậc cao như vậy "hợp với nhiễu" cũng như dữ liệu, và nói chung cho ra các kết quả tồi khi được dùng để tiên đoán y với các giá trị vô hình trước đó của x. Tốt hơn ta nên chọn n nhỏ hơn đáng kể so với m và hy vọng bằng cách chọn tốt các hệ số c_j, ta có thể được một hàm F tìm các khuôn mẫu quan trọng trong các điểm dữ liệu mà không tập trung quá đáng vào nhiễu. Có vài nguyên tắc lý thuyết tồn tại để chọn n, nhưng chúng vượt quá phạm vi của tài liệu này. Trong mọi trường hợp, sau khi chọn n, ta kết thúc bằng một tập hợp được xác định trên của các phương trình mà ta muốn giải cũng như có thể. Giờ đây, ta nêu cách thực hiện điều này.

Cho

$$A = \begin{pmatrix} f_1(x_1) & f_2(x_1) & \cdots & f_n(x_1) \\ f_1(x_1) & f_2(x_2) & \cdots & f_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_m) & f_2(x_m) & \cdots & f_n(x_m) \end{pmatrix}$$

thể hiện ma trận các giá trị của các hàm cơ sở tại các điểm đã cho; nghĩa là, $a_{ij} = f_j(x_i)$. Cho $c = (c_k)$ thể hiện vectơ các hệ số có kích cỡ n.

Thì,

$$Ac = \begin{pmatrix} f_1(x_1) & f_2(x_1) & \cdots & f_n(x_1) \\ f_1(x_2) & f_2(x_2) & \cdots & f_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_m) & f_2(x_m) & \cdots & f_n(x_m) \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix}$$

$$= \begin{pmatrix} F(x_1) \\ F(x_2) \\ \vdots \\ F(x_m) \end{pmatrix}$$

là vectơ kích cỡ-m của "các giá trị đã tiên đoán" với y. Như vậy, $\eta = Ac - y$ là vectơ kích cỡ-m của các ***lỗi xấp xỉ.***

Để giảm thiểu các lỗi xấp xỉ, ta quyết định giảm thiểu quy mẫu vectơ lỗi η, cho ta một ***giải pháp các bình phương bé nhất***, bởi

$$\|\eta\| = \left(\sum_{i=1}^{m} \eta_i^2\right)^{1/2}.$$

Bởi

$$||\eta||^2 = ||Ac - y||^2 = \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} c_j - y_i \right)^2 .$$

ta có thể giảm thiểu $||\eta||$ bằng cách lấy vi phân $||\eta||^2$ đối với mỗi c_k rồi ấn định kết quả là 0:

$$\frac{d||\eta||^2}{dc_k} = \sum_{i=1}^{m} 2\left(\sum_{j=1}^{n} a_{ij} c_j - y_i \right) a_{ik} = 0 . \qquad (31.32)$$

n phương trình (31.32) với $k = 1, 2, \ldots, n$ tương đương với phương trình ma trận đơn $(Ac - y)^T A = 0$ hoặc, theo tương đương (dùng Bài tập 31.1-3), với

$$A^T(Ac - y) = 0 ,$$

hàm ý

$$A^T Ac = A^T y . \qquad (31.33)$$

Trong thống kê học, điều này được gọi là ***phương trình chuẩn tắc***. Ma trận $A^T A$ là đối xứng theo Bài tập 31.1-3, và nếu A có hạng cột đầy đủ, thì $A^T A$ cũng là xác định dương. Do đó, $(A^T A)^{-1}$ tồn tại, và giải pháp cho phương trình (31.33) là

$$\begin{aligned} c &= ((A^T A)^{-1} A^T) y \\ &= A^+ y , \end{aligned} \qquad (31.34)$$

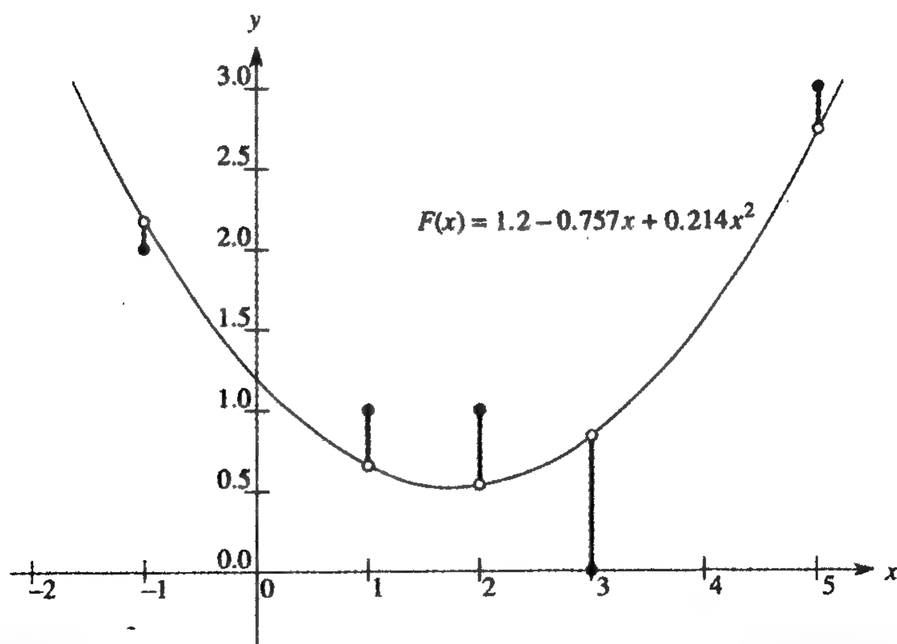
ở đó ma trận $A^+ = ((A^T A)^{-1} A^T)$ được gọi là ***phép đảo giả*** [pseudoinverse] của ma trận A. Phép đảo giả là một phép tổng quát hóa tự nhiên của khái niệm của một ma trận nghịch đảo với trường hợp ở đó A không bình phương. (So sánh phương trình (31.34) như giải pháp xấp xỉ cho $Ac = y$ với giải pháp $A^{-1}b$ như giải pháp chính xác của $Ax = b$.)

Để lấy ví dụ về việc tạo một san bằng theo các bình phương bé nhất, giả sử rằng ta có 5 điểm dữ liệu

$$(-1, 2), (1, 1), (2, 1), (3, 0), (5, 3) ,$$

được nêu dưới dạng chấm đen trong Hình 31.3. Ta muốn san bằng các điểm bằng một đa thức bậc hai $F(x) = c_1 + c_2 x + c_3 x^2$. Ta bắt đầu bằng ma trận của các giá trị hàm cơ bản

$$A = \begin{pmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \\ 1 & x_4 & x_4^2 \\ 1 & x_5 & x_5^2 \end{pmatrix} = \begin{pmatrix} 1 & -1 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \\ 1 & 5 & 25 \end{pmatrix} ,$$



Hình 31.3 San bằng theo các bình phương bé nhất của một đa thức bậc hai theo tập hợp các điểm dữ liệu $\{(-1, 2), (1, 1), (2, 1), (3, 0), (5, 3)\}$. Các chấm đen là các điểm dữ liệu, và các chấm trắng là các giá trị dự trù của chúng mà đa thức tiên đoán $F(x) = 1.2 - 0.757x + 0.214x^2$, đa thức bậc hai giảm thiểu tổng của các lỗi bình phương. Lỗi cho từng điểm dữ liệu được nêu dưới dạng một vạch tô bóng.

phép đảo giả của nó là

$$A^+ = \begin{pmatrix} 0.500 & 0.300 & 0.200 & 0.100 & -0.100 \\ -0.388 & 0.093 & 0.190 & 0.193 & -0.088 \\ 0.060 & -0.036 & -0.048 & -0.036 & 0.060 \end{pmatrix}.$$

Nhân y với A^+, ta được vectơ hệ số

$$c = \begin{pmatrix} 1.200 \\ -0.757 \\ 0.214 \end{pmatrix},$$

tương ứng với đa thức bậc hai $F(x) = 1.200 - 0.757x + 0.214x^2$

dưới dạng bậc hai vẽ theo các điểm sát nhất đối với dữ liệu đã cho, theo nghĩa các bình phương bé nhất.

Là một vấn đề thực tiễn, ta giải phương trình chuẩn tắc (31.33) bằng cách nhân y với A^T rồi tìm một phân tích LU của A^TA. Nếu A có hạng đầy đủ, ma trận A^TA được bảo đảm là phi lập dị, bởi nó đối xứng và xác định dương. (Xem Bài tập 31.1-3 và Định lý 31.6.)

Bài tập

31.6-1

Chứng minh mọi đường chéo thành phần của một ma trận xác định dương đối xứng là dương.

31.6-2

Cho $A = \begin{pmatrix} a & b \\ b & c \end{pmatrix}$ là một ma trận xác định dương đối xứng 2 x 2. Chứng minh định thức của nó $ac - b^2$ là dương bằng cách "hoàn tất bình phương" theo cách tương tự như đã được dùng trong phần chứng minh của Bổ đề 31.15.

31.6-3

Chứng minh thành phần cực đại trong một ma trận xác định dương đối xứng nằm trên đường chéo.

31.6-4

Chứng minh định thức của mỗi ma trận con dẫn đầu của một ma trận xác định dương đối xứng là dương.

31.6-5

Cho A_k thể hiện ma trận con dẫn đầu thứ k của một ma trận xác định dương đối xứng A. Chứng minh $\det(A_k)/\det(A_{k-1})$ là trục quay thứ k trong khi chạy phân tích LU, ở đó theo quy ước $\det(A_0) = 1$.

31.6-6

Tìm hàm có dạng

$F(x) = c_1 + c_2 x \lg x + c_3 e^x$

nghĩa là san bằng tốt nhất theo các bình phương bé nhất đối với các điểm dữ liệu.

(1,1), (2,1), (3,3), (4,8) .

31.6-7

Chứng tỏ phép đảo giả A^+ thỏa bốn phương trình sau:

$$AA^+A = A ,$$

$$A^+AA^+ = A^+ ,$$

$$(AA^+)^T = AA^+ ,$$

$$(A^+A)^T = A^+A .$$

Các Bài Toán

31-1 Thuật toán phép nhân ma trận bool Shamir

Trong Đoạn 31.3, ta đã nhận thấy thuật toán phép nhân ma trận Strassen không thể áp dụng trực tiếp cho phép nhân ma trận bool bởi vì tựa vành bool $Q = (\{0, 1\}, \vee, \wedge, 0, 1)$ không phải là một vành. Tuy nhiên, Định lý 31.10 đã chứng tỏ nếu ta dùng các phép toán số học trên các từ $O(\lg n)$ bit, ta vẫn có thể áp dụng phương pháp Strassen để nhân các ma trận bool $n \times n$ trong $O(n^{\lg 7})$ thời gian. Trong bài toán này, ta nghiên cứu một phương pháp xác suất chỉ dùng các phép toán bit để đạt được một cận tốt gần như vậy nhưng với một cơ may bị lỗi nhỏ.

a. Chứng tỏ $R = (\{0,1\}, \oplus, \wedge, 0, 1)$, ở đó $\oplus$ là hàm XOR (hoặc loại trừ), là một vành.

Cho $A = (a_{ij})$ và $B = (b_{ij})$ là các ma trận bool $n \times n$, và cho $C = (c_{ij}) = AB$ trong tựa vành Q. Phát sinh $A' = (a'_{ij})$ từ A dùng thủ tục ngẫu nhiên hóa dưới đây:

- Nếu $a_{ij} = 0$, thì cho $a'_{ij} = 0$
- Nếu $A_{ij} = 1$, thì cho $a'_{ij} = 1$ với xác suất 1/2 và cho $a'_{ij} = 0$ với xác suất 1/2. Các chọn lựa ngẫu nhiên cho mỗi khoản nhập là bậcc lập.

b. Cho $C' = (c'_{ij}) = A'B$ trong vành R. Chứng tỏ $c_{ij} = 0$ hàm ý $c_{ij} = 0$. Chứng tỏ $c_{ij} = 1$ hàm ý $c'_{ij} = 1$ với xác suất 1/2.

c. Chứng tỏ với bất kỳ $\varepsilon > 0$, xác suất tối đa là ε/n^2 mà một c'_{ij} đã cho không bao giờ nhận giá trị c_{ij} cho $\lg(n^2/\varepsilon)$ chọn lựa bậcc lập của ma trận A'. Chứng tỏ rằng xác suất ít nhất là $1 - \varepsilon$ mà tất cả c'_{ij} nhận các giá trị đúng đắn của chúng ít nhất một lần.

d. Nêu một thuật toán ngẫu nhiên hóa $O(n^{\lg 7} \lg n)$-thời gian tính toán tích trong tựa vành bool Q của hai ma trận $n \times n$ với xác suất ít nhất $1 - 1/n^k$ với một hằng $k > 0$. Các phép toán duy nhất được phép trên các thành phần ma trận là $\vee$, $\wedge$, và $\oplus$.

31-2 Các hệ thống ba đường chéo của các phương trình tuyến tính

Xét ma trận ba đường chéo

$$A = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{pmatrix}.$$

a. Tìm một phân tích LU của A.

b. Giải quyết phương trình $Ax = (\ 1\ 1\ 1\ 1\ 1\ )^T$ bằng cách dùng phép thay tới và lui.

c. Tìm nghịch đảo của A.

d. Chứng tỏ với bất kỳ ma trận $n \times n$ đối xứng, xác định dương, ba đường chéo, A và bất kỳ n-vectơ b, phương trình $Ax = b$ có thể được giải quyết trong $O(n)$ thời gian bằng cách thực hiện một phân tích LU. Chứng tỏ mọi phương pháp dựa trên việc hình thành A^{-1} theo tiệm cận đều tốn kém hơn trong ca xấu nhất.

e. Chứng tỏ với bất kỳ ma trận $n \times n$ phi lập dị, ba đường chéo, A và bất kỳ n-vector b, phương trình $Ax = b$ có thể được giải quyết trong $O(n)$ thời gian bằng cách thực hiện một phân tích LUP.

31-3 Các hàm splin

Một phương pháp thực tiễn để nội suy một tập hợp các điểm với một đường cong đó là sử dụng các ***hàm splin bậc ba***. Ta có một tập hợp $\{(x_i, y_i) : i = 0, 1, ..., n\}$ gồm $n + 1$ cặp giá trị điểm, ở đó $x_0 < x_1 < ... < x_n$. Ta muốn vẽ một đường cong (chốt trục) bậc ba từng khúc $f(x)$ theo các điểm. Nghĩa là, đường cong $f(x)$ được hình thành bởi n đa thức bậc ba $f_i(x) = a_i + b_i x + c_i x^2 + d_i x^3$ với $i = 0,1,..., n - 1$, ở đó nếu x rơi vào miền $x_i \le x \le x_{i+1}$ thì giá trị của đường cong được căn cứ vào $f(x) = f_i(x - x_i)$. Các điểm x_i tại đó các đa thức bậc ba được "dán" với nhau được gọi là các ***nút*** [knots]. Để đơn giản, ta mặc nhận rằng $x_i = i$ với $i = 0, 1, ..., n$.

Để bảo đảm tính liên tục của $f(x)$, ta yêu cầu

$$f(x_i) = f_i(0) = y_i\ ,$$
$$f(x_{i+1}) = f_i(1) = y_{i+1}$$

với $i = 0,1,..., n - 1$. Để bảo đảm $f(x)$ đủ trơn, ta cũng nhấn mạnh phải có tính liên tục của đạo hàm đầu tiên tại mỗi nút:

$$f'(x_{i+1}) = f'_i(1) = f'_{i+1}(0)$$

với $i = 0, 1, ..., n - 1$.

a. Giả sử rằng với $i = 0, 1, ..., n$, ta không những có các cặp giá trị-điểm $\{(x_i, y_i)\}$ mà còn các đạo hàm đầu tiên $D_i = f'(x_i)$ tại mỗi nút. Diễn tả mỗi hệ số a_i, b_i, c_i và d_i theo dạng các giá trị y_i, y_{i+1}, D_i, và D_{i+1}. (Nên nhớ $x_i = i$.) $4n$ hệ số có thể tính toán nhanh đến mức nào từ các cặp giá trị điểm và các đạo hàm đầu tiên?

Vấn đề còn lại đó là cách chọn các đạo hàm đầu tiên của $f(x)$ tại các nút. Một phương pháp đó là yêu cầu các đạo hàm thứ hai tiếp tục tại các nút:

$f''(x_{i+1}) = f''_i(1) = f''_{i+1}(0)$

với $i = 0, 1, ..., n - 1$. Tại các nút đầu và cuối, ta mặc nhận $f''(x_0) = f''_0(0) = 0$ và $f''(x_n) = f''_n(1) = 0$; các giả thiết này khiến $f(x)$ trở thành một ***hàm splin bậc ba tự nhiên***.

b. Dùng các hạn chế về tính liên tục trên đạo hàm thứ hai để chứng tỏ với $i = 1, 2, ..., n - 1$,

$$D_{i-1} + 4D_i + D_{i+1} = 3(y_{i+1} - y_{i-1}) \qquad (31.35)$$

c. Chứng tỏ

$$2D_0 + D_1 = 3(y_1 - y_0) \qquad (31.36)$$

$$D_{n-1} + 2D_n = 3(y_n - y_{n-1}) . \qquad (31.37)$$

d. Viết lại các phương trình (31.35)-(31.37) dưới dạng một phương trình ma trận bao hàm vectơ $D = \langle D_0, D_1, ..., D_n \rangle$ của các ẩn số. Ma trận trong phương trình của bạn có các thuộc tính nào?

e. Chứng tỏ một tập hợp $n + 1$ cặp giá trị điểm có thể nội suy bằng một hàm splin bậc ba tự nhiên trong $O(n)$ thời gian (xem Bài toán 31-2).

f. Nêu cách xác định một hàm splin bậc ba tự nhiên nội suy một tập hợp $n + 1$ điểm (x_i, y_i) thỏa $x_0 < x_1 < ... < x_n$, thậm chí khi x_i không nhất thiết bằng với i. Phải giải phương trình ma trận nào, và thuật toán của bạn chạy nhanh tới mức nào?

Ghi chú Chương

Có nhiều tài liệu tuyệt vời sẵn có mô tả phép tính số và khoa học chi tiết hơn nhiều so với ở đây. Dưới đây là những nội dung nên đọc: George và Liu [81], Golub và Van Loan [89], Press, Flannery, Teukolsky, và Vetterling [161, 162], và Strang [181, 182].

Ấn bản của thuật toán Strassen vào năm 1969 [183] đã gây nhiều náo bậcng. Nhưng trước đó, người ta khó lòng tưởng tượng rằng lại có thể cải thiện thuật toán đơn sơ. Từ đó, cận trên tiệm cận trên sự khó khăn của phép nhân ma trận đã được cải thiện đáng kể. Thuật toán hiệu quả nhất theo tiệm cận để nhân các ma trận $n \times n$ hiện nay, của Coppersmith và Winograd [52], có một thời gian thực hiện là $O(n^{2.376})$. Phần trình bày đồ họa của thuật toán Strassen là của Paterson [155]. Fischer và Meyer [67] đã thích ứng thuật toán Strassen cho các ma trận bool (Định lý 31.10).

Phép khử Gauss, là cơ sở cho các phân tích LU và LUP, là phương pháp có hệ thống đầu tiên để giải các hệ thống các phương trình tuyến

tính. Nó cũng là một trong các thuật toán số sớm nhất. Mặc dù đã được biết đến trước đó, người ta thường quy sự khám phá của nó cho *C.* F. Gauss (1777-1855). Trong tài liệu nổi tiếng của mình [183], Strassen cũng đã chứng tỏ một ma trận $n \times n$ có thể được đảo trong $O(n^{\lg 7})$ thời gian. Winograd [203] ban đầu đã chứng minh phép nhân ma trận không khó hơn phép nghịch đảo ma trận, và đảo đề là của Aho, Hopcroft, và Ullman [4].

Strang [182] có một phần trình bày tuyệt vời về các ma trận xác định dương đối xứng và về đại số học tuyến tính nói chung. Ông đã có chú thích dưới đây trên trang 334: "Lớp tôi thường hỏi về các ma trận xác định dương *phi đối xứng* [unsymmetric]. Tôi không bao giờ dùng thuật ngữ này."

32 Các Đa Thức và FFT

Phương pháp đơn giản của cộng hai đa thức có bậc *n* chiếm $\Theta(n)$ thời gian, nhưng phương pháp đơn giản để nhân chiếm $\Theta(n^2)$ thời gian. Trong chương này, ta sẽ nêu cách thức mà Fast Fourier Transform, hoặc FFT, có thể rút gọn thời gian để nhân các đa thức còn $O(n \lg n)$.

Các đa thức

Một ***đa thức*** trong biến x trên một trường đại số F là một hàm $A(x)$ có thể được biểu diễn như sau:

$$A(x) = \sum_{j=0}^{n-1} a_j x^j .$$

Ta gọi n là ***cận bậc*** [degree-bound] của đa thức, và gọi các giá trị a_0, $a_1, \ldots, a_{n-1}$ là các ***hệ số*** của đa thức. Các hệ số được rút từ trường F, thường là tập hợp **C** các số phức. Một đa thức $A(x)$ được gọi là có ***bậc*** k nếu hệ số phi zero cao nhất của nó là a_k. Bậc của một đa thức có cận bậc n có thể là bất kỳ số nguyên nào giữa 0 và $n - 1$, kể cả các số này. Ngược lại, một đa thức có bậc k là một đa thức có cận bậc n với bất kỳ $n > k$.

Có nhiều phép toán khác nhau mà ta có thể muốn định nghĩa cho các đa thức. Với *phép cộng đa thức*, nếu $A(x)$ và $B(x)$ là các đa thức có cận bậc n, ta nói rằng ***tổng*** của chúng là một đa thức $C(x)$, cũng có cận bậc n, sao cho $C(x) = A(x) + B(x)$ với tất cả x trong trường cơ sở. Nghĩa là, nếu $n - 1$

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

và

$$B(x) = \sum_{j=0}^{n-1} b_j x^j ,$$

thì

$$C(x) = \sum_{j=0}^{n-1} c_j x^j ,$$

ở đó $c_j = a_j + b_j$ với $j = 0, 1, \ldots, n - 1$. Ví dụ, nếu $A(x) = 6x^3 + 7x^2 - 10x$

$+ 9$ và $B(x) = -2x^3 + 4x - 5$, thì $C(x) = 4x^3 + 7x^2 - 6x + 4$.

Với ***phép nhân đa thức***, nếu $A(x)$ và $B(x)$ là các đa thức có cận bậc n, ta nói rằng ***tích*** $C(x)$ của chúng là một đa thức có cận bậc $2n - 1$ sao cho $C(x) = A(x)B(x)$ với tất cả x trong trường cơ sở. Bạn ắt đã từng nhân các đa thức, bằng cách nhân mỗi số hạng trong $A(x)$ với mỗi số hạng trong $B(x)$ và tổ hợp các số hiệu với các lũy thừa bằng. Ví dụ, ta có thể nhân $A(x) = 6x^3 + 7x^2 - 10x + 9$ và $B(x) = -2x^3 + 4x - 5$ như sau:

$$
\begin{array}{rrrrrrr}
 & & & 6x^3 & +\ 7x^2 & -\ 10x & +\ 9 \\
 & & & -\ 2x^3 & & +\ 4x & -\ 5 \\
\hline
 & & & -\ 30x^3 & -\ 35x^2 & +\ 50x & -\ 45 \\
 & & 24x^4 & +\ 28x^3 & -\ 40x^2 & +\ 36x & \\
-\ 12x^6 & -\ 14x^5 & +\ 20x^4 & -\ 18x^3 & & & \\
\hline
-\ 12x^6 & -\ 14x^5 & +\ 44x^4 & -\ 20x^3 & -\ 75x^2 & +\ 86x & -\ 45
\end{array}
$$

Một cách khác để diễn tả tích $C(x)$ đó là

$$C(x) = \sum_{j=0}^{2n-1} c_j x^j , \qquad (32.1)$$

ở đó

$$c_j = \sum_{k=0}^{j} a_k b_{j-k} . \qquad (32.2)$$

Lưu ý, bậc(C) = bậc(A) + bậc(B), ngụ ý

cận bậc(C) = cận bậc(A) + cận bậc$(B) - 1$

$\leq$ cận bậc(A) + cận bậc(B) .

Tuy nhiên, ta sẽ nói về cận bậc của C như là tổng các cận bậc của A và B, bởi nếu một đa thức có cận bậc k nó cũng có cận bậc $k + 1$.

Khái quát chương

Đoạn 32.1 trình bày hai cách để biểu diễn các đa thức: phần biểu diễn hệ số và phần biểu diễn giá trị điểm. Các phương pháp đơn giản để nhân các đa thức—các phương trình (32.1) và (32.2)—chiếm $\Theta(n^2)$ thời gian khi các đa thức được biểu thị theo dạng hệ số, nhưng chỉ $\Theta(n)$ thời gian khi chúng được biểu thị theo dạng giá trị điểm. Tuy nhiên, ta có thể nhân các đa thức dùng phần biểu diễn hệ số trong chỉ $\Theta(n \lg n)$ thời gian bằng cách chuyển đổi giữa hai phép biểu diễn. Để xem tại sao điều này làm việc, trước tiên ta phải nghiên cứu các căn phức của đơn vị, mà ta thực hiện trong Đoạn 32.2. Sau đó, ta dùng FFT và nghịch đảo củanó, cũng được mô tả trong Đoạn 32.2, để thực hiện các phép chuyển

đổi. Đoạn 32.3 có nêu cách thực thi FFT nhanh chóng trong cả hai mô hình nối tiếp và song song.

Chương này chủ yếu sử dụng các số phức, và ký hiệu i sẽ được dùng rộng rãi để thể hiện $\sqrt{-1}$.

32.1 Phần biểu diễn của các đa thức

Theo một nghĩa nào đó, các phép biểu diễn hệ số và giá trị điểm của các đa thức là tương đương; nghĩa là, một đa thức theo dạng giá trị điểm sẽ có một đối tác duy nhất theo dạng hệ số. Trong đoạn này, ta giới thiệu hai phép biểu diễn và nêu cách tổ hợp chúng để cho phép tiến hành nhân hai đa thức có cận bậc n trong $\Theta(n \lg n)$ thời gian.

Phần biểu diễn hệ số

Một ***phần biểu diễn hệ số*** của một đa thức $A(x) = \sum_{j=0}^{n-1} a_j x^j$ có cận bậc n là một vectơ các hệ số $a = (a_0, a_1, ..., a_{n-1})$. Trong các phương trình ma trận trong chương này, nói chung ta sẽ xem các vectơ như các vectơ cột.

Phần biểu diễn hệ số tỏ ra tiện dụng đối với một số các phép toán trên các đa thức. Ví dụ, phép toán ***đánh giá*** đa thức $A(x)$ tại một điểm x_0 đã cho bao gồm việc tính toán giá trị của $A(x_0)$. Việc đánh giá mất một thời gian $\Theta(n)$ dùng ***quy tắc Homer:***

$$A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + ... + x_0(a_{n-2} + x_0(a_{n-1}))...)) .$$

Cũng vậy, phép cộng hai đa thức được biểu diễn bằng các vectơ hệ số $a = (a_0, a_1, ..., a_{n-1})$ và $b = (b_0, b_1, ..., b_{n-1})$ mất $\Theta(n)$ thời gian: ta chỉ kết xuất vectơ hệ số $c = (c_0, c_1, ..., c_{n-1})$, ở đó $c_j = a_j + b_j$ với $j = 0, 1, ..., n-1$.

Giờ đây, xét phép nhân của hai đa thức $A(x)$ và $B(x)$ có cận bậc n được biểu diễn theo dạng hệ số. Nếu ta dùng phương pháp mà các phương trình (32.1) và (32.2) mô tả, phép nhân đa thức mất một thời gian $\Theta(n^2)$, bởi mỗi hệ số trong vectơ a phải được nhân với mỗi hệ số trong vectơ b. Phép toán nhân các đa thức theo dạng hệ số dường như khó hơn đáng kể so với trường hợp đánh giá một đa thức hoặc cộng hai đa thức. Vectơ hệ số kết quả c, căn cứ vào phương trình (32.2), còn được gọi là ***phép tích chập*** [convolution] của các vectơ đầu vào a và b, được thể hiện $c = a \otimes b$. Bởi việc nhân các đa thức và tính toán các phép tích chập là các bài toán tính toán căn bản có tầm thực tiễn đáng kể, nên chương này tập trung vào các thuật toán hiệu quả cho chúng.

32.1 *Phần biểu diễn của các đa thức*

Phần biểu diễn giá trị điểm

Một ***phần biểu diễn giá trị điểm*** của một đa thức $A(x)$ có cận bậc n là một tập hợp ***n cặp giá trị điểm*** [point- value pairs]

$\{(x_0, y_0), (x_1, y_1), ..., (x_{n-1}, y_{n-1})\}$ sao cho tất cả x_k đều riêng biệt và

$$y_k = A(x_k) \tag{32.3}$$

với $k = 0, 1,..., n - 1$. Một đa thức có nhiều phép biểu diễn giá trị điểm, bởi mọi tập hợp n điểm riêng biệt $x_0, x_1, ..., x_{n-1}$ có thể được dùng làm cơ sở cho phần biểu diễn.

Việc tính toán một phần biểu diễn giá trị điểm cho một đa thức đã cho theo dạng hệ số về nguyên tắc là đơn giản, bởi ta chỉ cần lựa n điểm riêng biệt $x_0, x_1,..., x_{n-1}$ rồi đánh giá $A(x_k)$ với $k = 0, 1,..., n - 1$. Với phương pháp Horner, tiến trình đánh giá n điểm này mất một thời gian $\Theta(n^2)$. Rồi đây ta sẽ thấy rằng nếu chọn *xk* một cách thông minh, ta có thể gia tốc phép tính này để chạy trong thời gian $\Theta(n \lg n)$.

Nghịch đảo của phép đánh giá—xác định dạng hệ số của một đa thức từ một phần biểu diễn giá trị điểm—được gọi là ***phép nội suy***. Định lý dưới đây chứng tỏ phép nội suy được định nghĩa kỹ, giả định cận bậc của đa thức nội suy bằng số lượng các cặp giá trị điểm đã cho.

Định lý 32.1 (Tính duy nhất của một đa thức nội suy)

Với mọi tổ hợp $\{(x_0, y_0), (x_1, y_1), . . . , (x_{n-1}, y_{n-1})\}$ gồm n cặp giá trị điểm, ta có một đa thức duy nhất $A(x)$ có cận bậc n sao cho $y_k = A(x_k)$ với $k= 0, 1,..., n - 1$.

Chứng minh Phần chứng minh dựa trên sự tồn tại của nghịch đảo của một số ma trận nhất định. Phương trình (32.3) tương đương với phương trình ma trận

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix} \tag{32.4}$$

Ma trận bên trái được ký hiệu là $V(x_0, x_1,..., x_{n-1})$ và được xem là một ma trận Vandermonde. Theo Bài tập 31.1-10, ma trận này có định thức

$$\prod_{j<k}(x_k - x_j) \ ,$$

và do đó, theo Định lý 31.5, nó là đảo nghịch (nghĩa là, phi lập dị)

nếu x_k là riêng biệt. Như vậy, các hệ số a_j có thể được giải với duy nhất phần biểu diễn giá trị điểm:

$$a = V(x_0, x_1, \ldots, x_{n-1})^{-1} y \; .$$

Phần chứng minh của Định lý 32.1 mô tả một thuật toán cho phép nội suy dựa trên việc giải tập hợp (32.4) các phương trình tuyến tính. Dùng phân tích LU các thuật toán của Chương 31, ta có thể giải các phương trình này trong thời gian $O(n^3)$.

Một thuật toán nhanh hơn cho phép nội suy n-điểm dựa trên ***công thức Lagrange:***

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \neq k} (x - x_j)}{\prod_{j \neq k} (x_k - x_j)} \; . \qquad (32.5)$$

Bạn có thể muốn xác minh phía bên phải của phương trình (32.5) là một đa thức có cận bậc n thỏa $A(x_k) = y_k$ với tất cả k. Bài tập 32.1-4 yêu cầu bạn nêu tính toán các hệ số của A dùng công thức Lagrange trong thời gian $\Theta(n^2)$.

Như vậy, phép đánh giá n-điểm và phép nội suy là các phép toán nghịch đảo được định nghĩa kỹ biến đổi giữa phần biểu diễn hệ số của một đa thức và một phần biểu diễn giá trị điểm.[1] Các thuật toán mô tả trên đây cho các bài toán này chiếm thời gian $\Theta(n^2)$.

Phần biểu diễn giá trị điểm khá tiện dụng cho nhiều phép toán trên các đa thức. Với phép cộng, nếu $C(x) = A(x) + B(x)$, thì $C(x_k) = A(x_k) + B(x_k)$ với bất kỳ điểm x_k. Chính xác hơn, nếu ta có một phần biểu diễn giá trị điểm cho A,

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\} \; ,$$

và cho B,

$$\{(x_0, y'_0), (x_1, y'_1), \ldots, (x_{n-1}, y'_{n-1})\}$$

(lưu ý A và B được đánh giá tại cùng n điểm), thì một phần biểu diễn giá trị điểm cho C là

$$\{(x_0, y_0 + y'_0), (x_1, y_1 + y'_1), \ldots, (x_{n-1}, y_{n-1} + y'_{n-1}\} \; .$$

Như vậy, thời gian để cộng hai đa thức có cận bậc n theo dạng giá trị điểm là $\Theta(n)$.

---

[1] Phép nội suy là một bài toán nổi tiếng phức tạp trên quan điểm về tính ổn định số. Tuy các cách tiếp cận ở đây là đúng về toán học, song những khác biệt nhỏ trong các đầu vào hoặc các lỗi làm tròn trong phép tính có thể gây ra các khác biệt lớn về kết quả.

Cũng vậy, phần biểu diễn giá trị điểm tỏ ra tiện dụng để nhân các đa thức. Nếu $C(x) = A(x)B(x)$, thì $C(x_k) = A(x_k)B(x_k)$ với bất kỳ điểm x_k, và ta có thể theo từng điểm nhân một phần biểu diễn giá trị điểm của A với một phần biểu diễn giá trị điểm của B để được một phần biểu diễn giá trị điểm của C. Tuy nhiên, ta phải đối mặt với bài toán cận bậc của C là tổng các cận bậc của A và B. Một phần biểu diễn giá trị điểm chuẩn của A và B bao gồm n cặp giá trị điểm cho mỗi đa thức. Việc nhân chúng với nhau cho ta n cặp giá trị điểm của C, nhưng do cận bậc của C là $2n$, nên Định lý 32.1 hàm ý rằng ta cần $2n$ cặp giá trị điểm cho một phần biểu diễn giá trị điểm của C. Do đó, ta phải bắt đầu bằng các phần biểu diễn giá trị điểm "mở rộng" của A và của B bao gồm $2n$ cặp giá trị điểm mỗi phần biểu diễn. Cho một phần biểu diễn giá trị điểm mở rộng của A,

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{2n-1}, y_{2n-1})\} ,$$

và một phần biểu diễn giá trị điểm mở rộng tương ứng của B,

$$\{(x_0, y'_0), (x_1, y'_1), \ldots, (x_{2n-1}, y'_{2n-1})\} ,$$

thì một phần biểu diễn giá trị điểm của C là

$$\{(x_0, y_0 y'_0), (x_1, y_1 y'_1), \ldots, (x_{2n-1}, y_{2n-1} y'_{2n-1})\} .$$

Cho hai đa thức đầu vào theo dạng giá trị điểm mở rộng, ta thấy $\Theta(n)$ chính là thời gian để nhân chúng để có được dạng giá trị điểm của kết quả, nhỏ hơn nhiều so với thời gian cần thiết để nhân các đa thức theo dạng hệ số.

Cuối cùng, ta xét cách đánh giá một đa thức đã cho theo dạng giá trị điểm tại một điểm mới. Với bài toán này, hình như ta không có cách tiếp cận nào đơn giản hơn là chuyển đổi đa thức thành dạng hệ số trước, rồi sau đó đánh giá nó tại điểm mới.

Phép nhân nhanh của các đa thức theo dạng hệ số

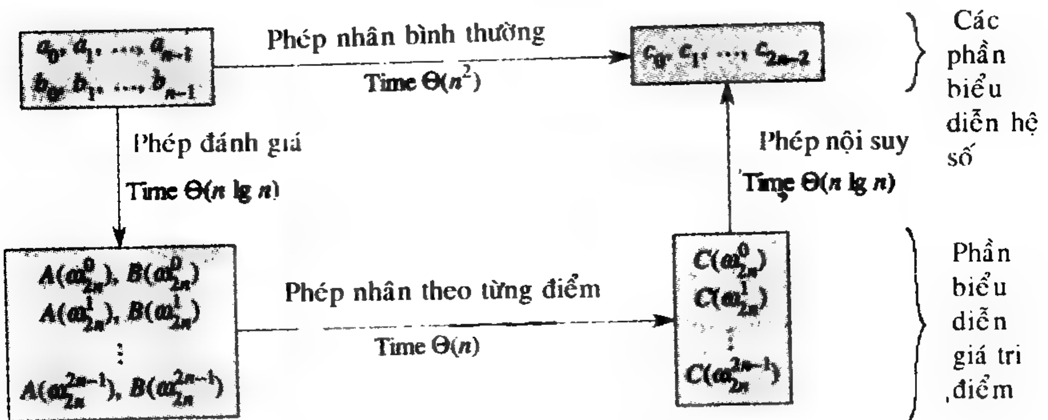
Có thể dùng phương pháp nhân thời gian tuyến tính cho các đa thức theo dạng giá trị điểm để giải quyết phép nhân đa thức theo dạng hệ số không? Câu trả lời xoay quanh khả năng của bạn trong việc chuyển đổi một đa thức nhanh chóng từ dạng hệ số thành dạng giá trị điểm (đánh giá) và ngược lại (nội suy).

Ta có thể dùng bất kỳ điểm mong muốn nào làm các điểm đánh giá, nhưng nhờ chọn các điểm đánh giá cẩn thận, ta có thể chuyển đổi giữa các phép biểu diễn trong chỉ $\Theta(n \lg n)$ thời gian. Như sẽ thấy trong Đoạn 32.2, nếu chọn "các căn phức của đơn vị" làm các điểm đáng giá, ta có thể tạo ra một phần biểu diễn giá trị điểm bằng cách lấy Discrete Fourier Transform (hoặc DFT) của một vectơ hệ số. Phép nghịch đảo, phép nội suy, có thể được thực hiện bằng cách lấy "DFT nghịch đảo" của các

cặp giá trị điểm, cho ra một vectơ hệ số. Đoạn 32.2 sẽ nêu cách FFT thực hiện các phép toán DFT và DFT nghịch đảo trong $\Theta(n \lg n)$ thời gian.

Hình 32.1 nêu chiến lược này theo đồ họa. Một chi tiết nhỏ liên quan đến các cận bậc. Tích của hai đa thức có cận bậc n là một đa thức có cận bậc $2n$. Do đó, trước khi đánh giá các đa thức đầu vào A và B, trước tiên ta nhân đôi các cận bậc của chúng thành $2n$ bằng cách cộng n hệ số thứ tự cao của 0. Bởi các vectơ có $2n$ thành phần, ta dùng "các căn phức thứ ($2n$) của đơn vị," được thể hiện bởi các số hạng ω_{2n} trong Hình 32.1.

Căn cứ vào FFT, ta có thủ tục $\Theta(n \lg n)$ thời gian dưới đây để nhân hai đa thức $A(x)$ và $B(x)$ có cận bậc n, ở đó các phép biểu diễn đầu vào và kết xuất theo dạng hệ số. Ta mặc nhận rằng n là một lũy thừa của 2; yêu cầu này có thể luôn được thỏa bằng cách cộng các hệ số zero thứ tự cao.



Hình 32.1 Một khái quát đồ họa của một tiến trình nhân đa thức hiệu quả. Các phần biểu diễn trên đầu theo dạng hệ số, trong khi các phần biểu diễn bên dưới theo dạng giá trị điểm. Các mũi tên từ trái qua phải tương ứng với phép nhân. Các số hạng ω_{2n} là các căn phức thứ $2n$ của đơn vị.

1. *Nhân đôi cận bậc*. Tạo các phần biểu diễn hệ số của $A(x)$ và $B(x)$ dưới dạng các đa thức cận bậc $2n$ bằng cách cộng n hệ số zero thứ tự cao vào mỗi đa thức.

2. *Đánh giá:* Tính toán các phần biểu diễn giá trị điểm của $A(x)$ và $B(x)$ có chiều dài $2n$ qua hai ứng dụng của FFT có thứ tự $2n$. Các phần biểu diễn này chứa các giá trị của hai đa thức tại các căn thứ ($2n$) của đơn vị.

3. *Nhân theo từng điểm*: Tính toán một phần biểu diễn giá trị điểm cho đa thức $C(x) = A(x)B(x)$ bằng cách nhân các giá trị này với nhau theo

từng điểm. Phép biểu diễn này chứa giá trị của $C(x)$ tại mỗi căn thứ 2n của đơn vị.

4. *Nội suy:* Tạo phần biểu diễn hệ số của đa thức $C(x)$ qua một ứng dụng đơn lẻ của một FFT trên $2n$ cặp giá trị điểm để tính toán DFT nghịch đảo.

Các bước (1) và (3) mất thời gian $\Theta(n)$, và các bước (2) và (4) mất thời gian $\Theta(n \lg n)$. Như vậy, một khi ta nêu cách sử dụng FFT, ta sẽ chứng minh như sau.

Định lý 32.2

Tích của hai đa thức có cận bậc n có thể được tính toán trong thời gian $\Theta(n \lg n)$, với cả hai phép biểu diễn đầu vào và kết xuất theo dạng hệ số.

Bài tập

32.1-1

Nhân các đa thức $A(x) = 7x^3 - x^2 + x - 10$ và $B(x) = 8x^3 - 6x + 3$ dùng các phương trình (32.1) và (32.2).

32.1-2

Cũng có thể tiến hành đánh giá một đa thức $A(x)$ có cận bậc n tại một điểm x_0 đã cho bằng cách chia $A(x)$ với đa thức $(x - x_0)$ để được một đa thức số thương $q(x)$ có cận bậc $n - 1$ và một số dư r, sao cho

$$A(x) = q(x)(x - x_0) + r \ .$$

Rõ ràng, $A(x_0) = r$. Nêu cách tính toán số dư r và các hệ số của $q(x)$ trong thời gian $\Theta(n)$ từ x_0 và các hệ số của A.

32.1-3

Suy ra một phần biểu diễn giá trị điểm với $A^{rev}(x) = \sum_{j=0}^{n-1} a_{n-1-j} x^j$ từ một phần biểu diễn giá trị điểm với $A(x) = \sum_{j=0}^{n-1} a_j x^j$, giả định không có điểm nào là 0.

32.1-4

Nêu cách sử dụng phương trình (32.5) để nội suy trong thời gian $\Theta(n^2)$. (*Mách nước:* Trước tiên tính toán $\Pi_j (x - x_k)$ và $\Pi_j (x_j - x_k)$ rồi chia với $(x - x_k)$ và $(x_j - x_k)$ theo yêu cầu với mỗi số hạng. Xem Bài tập 32.1-2.)

32.1-5

Giải thích những điểm sai với cách tiếp cận "hiển nhiên" đối với phép chia đa thức dùng một phần biểu diễn giá trị điểm. Mô tả tách biệt

trường hợp ở đó phép chia là chính xác và trường hợp ở đó nó không.

32.1-6

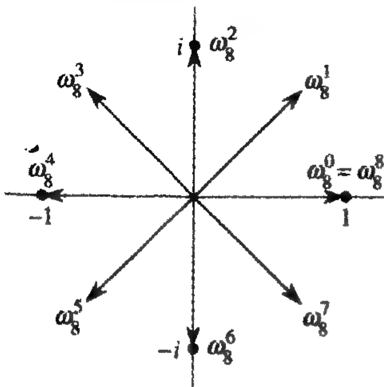
Xét hai tập hợp A và B, mỗi tập hợp có n số nguyên trong miền từ 0 đến $10n$. Ta muốn tính toán ***tổng Đề các*** của A và B, được định nghĩa bởi

$C = \{x + y : x \in A \text{ and } y \in B\}$.

Lưu ý, các số nguyên trong C nằm trong miền giá trị từ 0 đến $20n$. Ta muốn tìm ra các thành phần của C và số lần mà mỗi thành phần của C được thực hiện dưới dạng một tổng của các thành phần trong A và B. Chứng tỏ bài toán có thể được giải trong $O(n \lg n)$ thời gian. (*Mách nước:* Biểu diễn A và B dưới dạng các đa thức có bậc $10n$.)

32.2 DFT và FFT

Trong Đoạn 32.1, ta chứng minh nếu dùng các căn phức của đơn vị, ta có thể đánh giá và nội suy trong $\Theta(n \lg n)$ thời gian. Trong đoạn này, ta định nghĩa các căn phức của đơn vị và nghiên cứu các tính chất của chúng, định nghĩa DFT, rồi nêu cách thức mà FFT tính toán DFT và nghịch đảo củanó trong chỉ $\Theta(n \lg n)$ thời gian.



Hình 32.2 Các giá trị của $\omega_8^0, \omega_8^1, \ldots, \omega_8^7$ trong mặt phẳng phức, ở đó $\omega_8 = e^{2\pi i/8}$ là căn thứ 8 chính của đơn vị.

Các căn phức của đơn vị

Một ***căn phức thứ n của đơn vị*** là một số phức ω sao cho

$\omega^n = 1$.

Có chính xác n căn phức thứ n của đơn vị: chúng là $e^{2\pi ik/n}$ với $k = 0, 1, \ldots, n - 1$. Để diễn dịch công thức này, ta dùng phần định nghĩa của hàm mũ của một số phức:

$e^{iu} = \cos(u) + i \sin(u)$.

Hình 32.2 chứng tỏ n căn phức của đơn vị được định cách đều quanh vòng tròn với bán kính đơn vị có tâm tại gốc của mặt phẳng phức. Giá trị

$$\omega_n = e^{2\pi i/n} \qquad (32.6)$$

được gọi là ***căn thứ n chính của đơn vị*** [principal nth root of unity]; tất cả các căn phức thứ n của đơn vị khác đều là lũy thừa của ω_n.

n căn phức thứ n của đơn vị,

$$\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$$

hình thành một nhóm dưới phép nhân (xem Đoạn 33.3). Nhóm này có cùng cấu trúc như nhóm cộng tính $(\mathbf{Z}_n, +)$ modulo n, bởi $\omega_n^n = \omega_n^0 = 1$ hàm ý rằng $\omega_n^j \omega_n^k = \omega_n^{j+k} = \omega_n^{(j+k) \bmod n}$. Cũng vậy, $\omega_n^{-1} = \omega_n^{n-1}$. Các tính chất chủ yếu của các căn phức thứ n của đơn vị được cung cấp trong các bổ đề dưới đây.

Bổ đề 32.3 (Bổ đề khử)

Với bất kỳ số nguyên $n \geq 0$, $k \geq 0$, và $d > 0$,

$$\omega_{dn}^{dk} = \omega_n^k \qquad (32.7)$$

Chứng minh Bổ đề theo sau trực tiếp từ phương trình (32.6), bởi

$$\begin{aligned}\omega_{dn}^{dk} &= (e^{2\pi i/dn})^{dk} \\ &= (e^{2\pi i/n})^k \\ &= \omega_n^k .\end{aligned}$$

Hệ luận 32.4

Với bất kỳ số nguyên chẵn $n > 0$,

$$\omega_n^{n/2} = \omega_2 = -1 .$$

Chứng minh Phần chứng minh được để làm Bài tập 32.2-1.

Bổ đề 32.5 (Bổ đề chia đôi)

Nếu $n > 0$ là chẵn, thì các bình phương của n căn phức thứ n của đơn vị là $n/2$ căn phức thứ $(n/2)$ của đơn vị.

Chứng minh Theo bổ đề khử, ta có $(\omega_n^k)^2 = \omega_{n/2}^k$, với bất kỳ số nguyên không âm k. Lưu ý, nếu ta bình phương tất cả các căn phức thứ n của đơn vị, như vậy ta có mỗi căn thứ $(n/2)$ của đơn vị chính xác hai lần, bởi

$$\begin{aligned}(\omega^{k+n/2}) &= \omega_n^{2k+n} \\ &= \omega_n^{2k}\omega_n^n \\ &= \omega_n^{2k}\end{aligned}$$

$= (\omega_n^k)^2$.

Như vậy, ω_n^k và $\omega_n^{k+n/2}$ có cùng bình phương. Tính chất này cũng có thể được chứng minh bằng Hệ luận 32.4, bởi $\omega_n^{n/2} = -1$ hàm ý $\omega_n^{n/2} = -\omega_n^k$, và như vậy $(\omega_n^{k+n/2})^2 = (\omega_n^k)^2$.

Như sẽ thấy, bổ đề chia đôi là thiết yếu với cách tiếp cận chia để trị của chúng ta để chuyển đổi giữa hệ số và các phép biểu diễn giá trị điểm của các đa thức, bởi nó bảo đảm các bài toán con đệ quy chỉ lớn bằng phân nửa.

Bổ đề 32.6 (Bổ đề tổng)

Với bất kỳ số nguyên $n \geq 1$ và số nguyên không âm k không thể chia cho n,

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = 0 .$$

Chứng minh Bởi phương trình (3.3) áp dụng cho các giá trị phức,

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = \frac{(\omega_n^k)^j - 1}{\omega_n^k - 1}$$

$$= \frac{(\omega_n^n)^k - 1}{\omega_n^k - 1}$$

$$= \frac{(1)^k - 1}{\omega_n^k - 1}$$

$$= 0 .$$

Việc yêu cầu k không chia được cho n bảo đảm mẫu thức không phải là 0, bởi $\omega_n^k = 1$ chỉ khi k chia được cho n.

DFT

Hãy nhớ lại ta muốn đánh giá một đa thức

$$A(x) = \sum_{j=0}^{n-1} a_j x^j .$$

có cận bậc n tại $\omega_n^0, \omega_n^1, \omega_n^2, ..., \omega_n^{n-1}$ (nghĩa là, tại n căn phức thứ n của đơn vị).[2] Không để mất tính tổng quát, ta mặc nhận rằng n là một lũy thừa của 2, bởi một cận bậc đã cho có thể luôn được nâng lên—lúc

---

[2] Chiều dài n thực tế là nội dung mà ta đã tham chiếu dưới dạng $2n$ trong Đoạn 32.1, bởi ta nhân đôi cận độ của các đa thức đã cho trước khi đánh giá. Do đó, theo ngữ cảnh phép nhân đa thức, ta thực tế đang làm việc với các căn phức thứ $(2n)$ của đơn vị.

nào ta cũng có thể cộng các hệ số zero thứ tự cao mỗi khi cần. Ta mặc nhận rằng A được cung cấp theo dạng hệ số: $a = (a_0, a_1, \ldots, a_{n-1})$. Ta hãy định nghĩa các kết quả y_k, với $k = 0, 1, \ldots, n-1$, theo

$$\begin{aligned} y_k &= A(\omega_n^k) \\ &= \sum_{j=0}^{n-1} a_j \, \omega_n^{kj} . \end{aligned} \tag{32.8}$$

Vectơ $y = (y_0, y_1, \ldots, y_{n-1})$ là ***Discrete Fourier Transform* (*DFT*)** của vectơ hệ số $a = (a_0, a_1, \ldots, a_{n-1})$. Ta cũng viết $y = \mathrm{DFT}_n(a)$.

FFT

Nhờ dùng một phương pháp mệnh danh là *Fast Fourier Transform* (*FFT*), vận dụng các tính chất đặc biệt của các căn phức của đơn vị, ta có thể tính toán $\mathrm{DFT}_n(a)$ trong thời gian $\Theta(n \lg n)$, trái với $\Theta(n^2)$ thời gian của phương pháp đơn giản.

Phương pháp FFT sử dụng một chiến lược chia để trị, dùng các hệ số chỉ số chẵn và chỉ số lẻ của $A(x)$ tách biệt để định nghĩa hai đa thức mới có cận bậc $n/2$ $A^{[0]}(x)$ và $A^{[1]}(x)$:

$$A^{[0]}(x) = a_0 + a_2 x + a_4 x^2 + \ldots + a_{n-2} x^{n/2-1} ,$$
$$A^{[1]}(x) = a_1 + a_3 x + a_5 x^2 + \ldots + a_{n-1} x^{n/2-1} .$$

Lưu ý, $A^{[0]}$ chứa tất cả các hệ số chỉ số chẵn của A (phần biểu diễn nhị phân của chỉ số kết thúc trong 0) và $A^{[1]}$ chứa tất cả các hệ số chỉ số lẻ (phần biểu diễn nhị phân của chỉ số kết thúc trong 1). Như vậy dẫn đến

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2) , \tag{32.9}$$

sao cho bài toán đánh giá $A(x)$ tại $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$ rút gọn thành

1. đánh giá các đa thức có cận bậc $n/2$ $A^{[0]}(x)$ và $A^{[1]}(x)$ tại các điểm

$$(\omega_n^0)^2, (\omega_n^1)^2, \ldots, (\omega_n^{n-1})^2 \tag{32.10}$$

rồi

2. tổ hợp các kết quả theo phương trình (32.9).

Theo bổ đề chia đôi, danh sách các giá trị (32.10) không gộp n giá trị riêng biệt mà chỉ gộp $n/2$ căn phức thứ $n/2$ của đơn vị, với mỗi gốc xảy ra chính xác hai lần. Do đó, các đa thức $A^{[0]}$ và $A^{[1]}$ có cận bậc $n/2$ được đánh giá đệ quy tại $n/2$ căn phức thứ $n/2$ của đơn vị. Các bài toán con này chính xác cùng dạng như bài toán ban đầu, nhưng có nửa kích cỡ. Giờ đây, ta đã chia thành công một phép tính DFT_n thành phần thành hai phép tính $\mathrm{DFT}_{n/2}$ $n/2$ thành phần. Sự phân tích này là cơ sở cho thuật toán FFT đệ quy dưới đây, tính toán DFT của một vectơ n-thành phần $a = (a_0, a_1, \ldots, a_{n-1})$, ở đó n là một lũy thừa của 2.

RECURSIVE-FFT(a)

1 $n \leftarrow length[a]$ ▷ n là một lũy thừa của 2.

2 **if** $n = 1$

3 **then return** a

4 $\omega_n \leftarrow e^{2\pi i/n}$

5 $\omega \leftarrow 1$

6 $a^{[0]} \leftarrow (a_0, a_2, \dots, a_{n-2})$

7 $a^{[1]}$, $(a_1, a_3, \dots, a_{n-1})$

8 $y^{[0]} \leftarrow$ RECURSIVE-FFT($a^{[0]}$)

9 $y^{[1]} \leftarrow$ RECURSIVE-FFT($a^{[1]}$)

10 **for** $k \leftarrow 0$ **to** $n/2 - 1$

11 **do** $y_k \leftarrow y_k^{[0]} + \omega\, y_k^{[1]}$

12 $y_{k+(n/2)} \leftarrow y_k^{[0]} - \omega\, y_k^{[1]}$

13 $\omega \leftarrow \omega\, \omega_n$

14 **return** y ▷ y được mặc nhận là vectơ cột.

Thủ tục RECURSIVE-FFT làm việc như sau. Các dòng 2-3 biểu diễn cơ sở của đệ quy; DFT của một thành phần là chính thành phần đó, bởi trong trường hợp này

$$\begin{aligned} y_0 &= a_0 \omega_1^0 \\ &= a_0 \cdot 1 \\ &= a_0 . \end{aligned}$$

Các dòng 6-7 định nghĩa các vectơ hệ số với các đa thức $A^{[0]}$ và $A^{[1]}$. Các dòng 4, 5, và 13 bảo đảm ω được cập nhật đúng đắn sao cho mỗi khi các dòng 11-12 được thi hành, $\omega = \omega_n^k$. (Việc duy trì một giá trị liên tục của ω từ lần lặp lại này sang lần lặp lại khác sẽ tiết kiệm thời gian so với việc tính toán ω_n từ đầu mỗi lần qua vòng lặp **for**.) Các dòng 8-9 thực hiện các phép tính $\mathrm{DFT}_{n/2}$ đệ quy, ấn định, với $k = 0, 1, \dots, n/2 - 1$,

$$y_k^{[0]} = A^{[0]}(\omega_{n/2}^k) ,$$

$$y_k^{[1]} = A^{[1]}(\omega_{n/2}^k) ,$$

hoặc, bởi $\omega_{n/2}^k = \omega_n^{2k}$ theo bổ đề hủy,

$$y_k^{[0]} = A^{[0]}(\omega_n^{2k}) ,$$

$$y_k^{[1]} = A^{[1]}(\omega_n^{2k}) .$$

Các dòng 11-12 tổ hợp các kết quả của các phép tính $\mathrm{DFT}_{n/2}$ đệ quy. Với $y_0, y_1, \dots, y_{n/2-1}$, dòng 11 cho ra

$$\begin{aligned} y_k &= y_k^{[0]} + \omega_n^k y_k^{[1]} \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^k A^{[1]}(\omega_n^{2k}) \\ &= A(\omega_n^k) , \end{aligned}$$

ở đó dòng cuối của lập luận này là do phương trình (32.9). Với $y_{n/2}$, $y_{n/2+1}, \ldots, y_{n-1}$, cho $k = 0, 1, \ldots, n/2 - 1$, dòng 12 cho ra

$$\begin{aligned} y_{k+(n/2)} &= y_k^{[0]} - \omega_n^k y_k^{[1]} \\ &= y_k^{[0]} + \omega_n^{k+(n/2)} y_k^{[1]} \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k}) \\ &= A^{[0]}(\omega_n^{2k+n}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k+n}) \\ &= A(\omega_n^{k+(n/2)}) . \end{aligned}$$

Dòng thứ hai là do dòng đầu tiên bởi $\omega_n^{k+(n/2)} = -\omega_n^k$. Dòng thứ tư là do dòng thứ ba bởi $\omega_n^n = 1$ hàm ý $\omega_n^{2k} = \omega_n^{2k+n}$. Dòng cuối là do phương trình (32.9). Như vậy, vectơ y mà RECURSIVE-FFT trả về quả thực là DFT của vectơ đầu vào a.

Để xác định thời gian thực hiện của thủ tục RECURSIVE-FFT, ta lưu ý ngoại trừ các lệnh gọi đệ quy, mỗi lần triệu gọi mất một thời gian $\Theta(n)$, ở đó n là chiều dài của vectơ đầu vào. Do đó phép truy toán của thời gian thực hiện là

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \lg n) . \end{aligned}$$

Như vậy, ta có thể đánh giá một đa thức có cận bậc n tại các căn phức thứ n của đơn vị trong thời gian $\Theta(n \lg n)$ dùng Fast Fourier Transform [biến đổi Fourier nhanh].

Nội suy tại các căn phức của đơn vị

Giờ đây, ta hoàn tất lược đồ phép nhân đa thức bằng cách nêu cách nội suy các căn phức của đơn vị bằng một đa thức, cho phép ta chuyển đổi từ dạng giá trị điểm trở về dạng hệ số. Ta nội suy bằng cách viết DFT dưới dạng một phương trình ma trận rồi xem xét dạng của nghịch đảo ma trận.

Từ phương trình (32.4), ta có thể viết DFT dưới dạng tích ma trận $y = V_n a$, ở đó V_n là một ma trận Vandermonde chứa các lũy thừa thích hợp của ω_n:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} .$$

Khoản nhập (k, j) của V_n là ω_n^{kj} với $j, k = 0, 1, \ldots, n-1$, và các số mũ của các khoản nhập của V_n hình thành một bảng phép nhân.

Với phép toán nghịch đảo, mà ta viết dưới dạng $a = \mathrm{DFT}_n^{-1}(y)$, ta tiến hành bằng cách nhân y với ma trận V_n^{-1}, nghịch đảo của V_n.

Định lý 32.7

Với $j, k = 0, 1, \ldots, n-1$, khoản nhập (j, k) của V^{-1} is con ω_n^{-kj}/n.

Chứng minh Ta chứng tỏ $V_n^{-1}V_n = I_n$, ma trận đồng nhất thức $n \times n$. Xét khoản nhập (j, j') của $V_n^{-1} V_n$:

$$[V_n^{-1}V_n]_{jj'} = \sum_{k=0}^{n-1} (\omega_n^{-kj}/n)(\omega_n^{kj'})$$

$$= \sum_{k=0}^{n-1} \omega_n^{k(j'-j)}/n .$$

Phép lấy tổng này bằng 1 nếu $j' = j$, và bằng không nó là 0 theo bổ đề lấy tổng (Bổ đề 32.6). Lưu ý, ta dựa vào $-(n-1) < j' - j < n-1$, sao cho $j' - j$ không chia được với n, để có thể áp dụng bổ đề lấy tổng.

Cho ma trận nghịch đảo V_n^{-1}, ta có $\mathrm{DFT}_n^{-1}(y)$ dựa vào

$$a_j = \frac{1}{n}\sum_{k=0}^{n-1} y_k\,\omega_n^{-kj} \tag{32.11}$$

với $j = 0, 1, \ldots, n-1$. Bằng cách so sánh các phương trình (32.8) và (32.11), ta thấy nhờ sửa đổi thuật toán FFT để chuyển các vai trò của a và y, thay ω_n bằng ω_n^{-1}, và chia mỗi thành phần của kết quả cho n, ta tính toán DFT nghịch đảo (xem Bài tập 32.2-4). Như vậy, DFT^{-1} cũng có thể được tính toán trong $\Theta(n \lg n)$ thời gian.

Như vậy, nhờ dùng FFT và FFT nghịch đảo, ta có thể biến đổi một đa thức có cận bậc n tới lui giữa phần biểu diễn hệ số và một phần biểu diễn giá trị điểm của nó trong thời gian $\Theta(n \lg n)$. Trong ngữ cảnh phép nhân đa thức, ta đã nêu như sau.

Định lý 32.8 (Định lý nhân chập)

Với bất kỳ hai vectơ a và b có chiều dài n, ở đó n là một lũy thừa của 2,

$$a \otimes b = \mathrm{DFT}_{2n}^{-1}(\mathrm{DFT}_{2n}(a) \cdot \mathrm{DFT}_{2n}(b)) ,$$

ở đó các vectơ a và b được chêm bằng các 0 vào chiều dài $2n$ và - thể hiện tích theo từng thành phần của hai vectơ $2n$ thành phần.

Bài tập

32.2-1

Chứng minh Hệ luận 32.4.

32.2-2

Tính toán DFT của vectơ (0,1, 2, 3).

32.2-3

Làm Bài tập 32.1-1 bằng cách dùng lược đồ $\Theta(n \lg n)$ thời gian.

32.2-4

Viết mã giả để tính toán DFT_n^{-1} trong $\Theta(n \lg n)$ thời gian.

32.2-5

Mô tả phép tổng quát hóa của thủ tục FFT cho trường hợp ở đó n là một lũy thừa của 3. Nêu một phép truy toán cho thời gian thực hiện, và giải phép truy toán.

32.2-6*

Giả sử rằng thay vì thực hiện một FFT n-thành phần trên trường các số phức (ở đó n là chẵn), ta dùng vành $\mathbf{Z}_m$ của các số nguyên modulo m, ở đó $m = 2^{tn/2} + 1$ và t là một số nguyên dương tùy ý. Dùng $w = 2^t$ thay vì ω_n làm một căn thứ n chính của đơn vị, modulo m. Chứng minh DFT và DFT nghịch đảo được định nghĩa kỹ trong hệ thống này.

32.2-7

Cho một danh sách các giá trị $z_0, z_1, \ldots, z_{n-1}$ (có thể với các lần lặp lại), nêu cách tìm ra các hệ số của đa thức $P(x)$ có cận bậc n chỉ có các zero tại $z_0, z_1, \ldots, z_{n-1}$ (có thể với các lần lặp lại). Thủ tục của bạn sẽ chạy trong thời gian $O(n \lg^2 n)$. (*Mách nước:* Đa thức $P(x)$ có một zero tại z_j nếu và chỉ nếu $P(x)$ là một bội của $(x - z_j)$.)

32.2-8

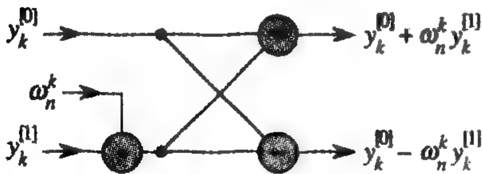
Phép biến đổi chirp của một vectơ $a = (a_0, a_1, \ldots, a_{n-1})$ là vectơ $y = (y_0, y_1, \ldots, y_{n-1})$, ở đó $y_k = \sum_{j=0}^{n-1} a_j z^{kj}$ và z là một số phức bất kỳ. Do đó, DFT là một trường hợp đặc biệt của phép biến đổi chirp, có được bằng cách lấy $z = \omega_n$. Chứng minh phép biến đổi chirp có thể được đánh giá trong thời gian $O(n \lg n)$ với một số phức z bất kỳ. (*Mách nước:* Dùng phương trình

$$y_k = z^{k^2/2} \sum_{j=0}^{n-1} \left(a_j z^{j^2/2}\right)\left(z^{-(k-j)^2/2}\right)$$

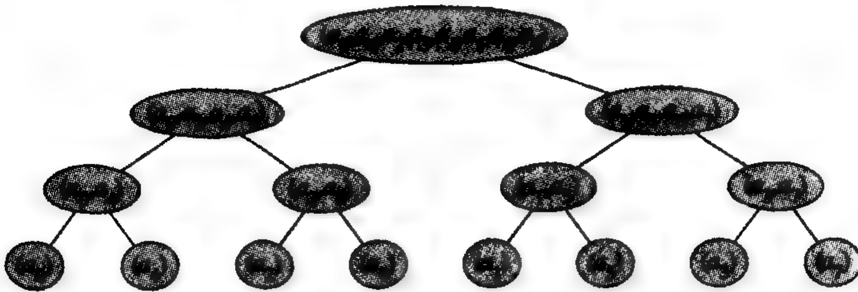
để xem phép biến đổi chirp như là một phép tích chập.)

32.3 Các thực thi FFT hiệu quả

Do các ứng dụng thực tiễn của DFT, như xử lý tín hiệu, yêu cầu tốc bậc tột cùng, đoạn này xét hai kiểu thực thi FFT hiệu quả. Trước hết, ta sẽ xét một phiên bản lặp lại của thuật toán FFT chạy trong $\Theta(n \lg n)$ thời gian nhưng có một hằng thấp hơn ẩn trong hệ ký hiệu Θ so với kiểu thực thi đệ quy trong Đoạn 32.2. Sau đó, ta dùng các hiểu biết sâu sắc đã dẫn ta đến cách thực thi lặp lại để thiết kế một mạch FFT song song hiệu quả.



Hình 32.3 Một phép toán bướm. Hai giá trị đầu vào nhập từ bên trái, ω_n^k được nhân với $y_k^{[1]}$, tổng và hiệu là kết xuất bên phải. Hình có thể được diễn dịch dưới dạng một mạch tổ hợp.



Hình 32.4 Cây các vectơ đầu vào cho các lệnh gọi đệ quy của thủ tục RECURSIVE-FFT. Lần triệu gọi ban đầu là với $n = 8$.

Một kiểu thực thi FFT lặp lại

Trước tiên ta lưu ý vòng lặp **for** của các dòng 10-13 của RECURSIVE-FFT liên quan đến việc tính toán giá trị $\omega_n^k y_k^{[1]}$ hai lần. Theo thuật ngữ bộ biên dịch, giá trị này được xem là một ***biểu thức con chung***. Ta có thể thay đổi vòng lặp để tính toán nó chỉ một lần, lưu trữ nó trong một biến tạm t.

for $k \leftarrow 0$ **to** $n/2 - 1$

  do $t \leftarrow \omega y_k^{[1]}$

    $y_k \leftarrow y_k^{[0]} + t$

    $y_{k+(n/2)} \leftarrow y_k^{[0]} - t$

$\omega \leftarrow \omega\, \omega_n$

Phép toán trong vòng lặp này, nhân ω (bằng với ω_n^k) với $y_k^{[1]}$, lưu trữ tích vào t, rồi cộng và trừ t ra khỏi $y_k^{[0]}$, được gọi là ***phép toán bướm*** [butterfly operation] và được nêu theo giản đồ trong Hình 32.3.

Giờ đây ta nêu cách làm cho thuật toán FFT lặp lại thay vì đệ quy trong cấu trúc. Trong Hình 32.4, ta đã dàn xếp các vectơ đầu vào theo các lệnh gọi đệ quy trong một lần triệu gọi của RECURSIVE-FFT trong một cấu trúc cây, ở đó lệnh gọi ban đầu là với $n = 8$. Cây có một mắt cho mỗi lệnh gọi của thủ tục, được gắn nhãn bởi vectơ đầu vào tương ứng. Mỗi lần triệu gọi RECURSIVE-FFT sẽ tạo hai lệnh gọi đệ quy, trừ phi nó đã nhận một vectơ 1 thành phần. Ta kiến lệnh gọi đầu tiên là con trái và lệnh gọi thứ hai là con phải.

Xem xét cây, ta nhận thấy nếu có thể dàn xếp các thành phần của vectơ ban đầu a vào thứ tự ở đó chúng xuất hiện trong các lá, ta có thể bắt chước việc thi hành của thủ tục RECURSIVE-FFT như sau. Trước tiên, ta lấy các thành phần trong các cặp, tính toán DFT của mỗi cặp bằng một phép toán bướm, và thay cặp bằng DFT của nó. Như vậy, vectơ lưu giữ $n/2$ DFT 2 thành phần. Kế đó, ta lấy $n/2$ DFT này trong các cặp và tính toán DFT của bốn thành phần vectơ mà chúng xuất phát bằng cách thi hành hai phép toán bướm, thay hai DFT 2-thành phần bằng một DFT 4-thành phần. Như vậy vectơ lưu giữ $n/4$ DFT 4-thành phần. Ta tiếp tục theo cách này cho đến khi vectơ lưu giữ hai DFT $(n/2)$ thành phần, mà ta có thể tổ hợp bằng $n/2$ phép toán bướm vào DFT n thành phần chung cuộc.

Để chuyển nhận xét này thành mã, ta dùng một mảng $A[0..n-1]$ mà thoạt đầu lưu giữ các thành phần của vectơ đầu vào a theo thứ tự mà chúng xuất hiện trong các lá của cây ở Hình 32.4. (Dưới đây ta sẽ nêu cách xác định thứ tự này.) Bởi tiến trình tổ hợp phải được thực hiện trên mỗi cấp của cây, nên ta giới thiệu một biến s để đếm các cấp, nằm trong phạm vi từ 1 (ở cuối, khi ta đang tổ hợp các cặp để tạo thành các DFT 2 thành phần) đến $\lg n$ (trên cùng, khi ta đang tổ hợp hai DFT $(n/2)$ thành phần để tạo ra kết quả chung cuộc). Do đó, thuật toán có cấu trúc dưới đây:

1 **for** $s \leftarrow 1$ **to** $\lg n$

2 **do for** $k \leftarrow 0$ **to** $n - 1$ **by** 2^s

3 **do** tổ hợp hai DFT 2^{s-1} thành phần trong
$A[k..\, k + 2^{s-1} - 1]$ và $A[k + 2^{s-1}..\, k + 2^s - 1]$
thành một DFT 2^s thành phần trong $A[k..\, k + 2^s - 1]$

Ta có thể diễn tả thân của vòng lặp (dòng 3) dưới dạng mã giả chính xác hơn. Ta chép vòng lặp **for** từ thủ tục RECURSIVE-FFT, định danh $y^{[0]}$ bằng $A[k \,..\, k + 2^{s-1} - 1]$ và $y^{[1]}$ bằng $A[k + 2^{s-1} \,..\, k + 2^s - 1]$. Giá trị của ω được dùng trong mỗi phép toán bướm tùy thuộc vào giá trị của s; ta dùng ω_m, ở đó $m = 2^s$. (Ta giới thiệu biến m chỉ với mục đích dễ đọc.) Ta giới thiệu một biến tạm khác u cho phép ta thực hiện phép toán bướm tại chỗ. Khi ta thay dòng 3 của cấu trúc chung bằng thân vòng lặp, ta có mã giả dưới đây, hình thành cơ sở của thuật toán FFT lặp lại cuối cùng của chúng ta cũng như kiểu thực thi song song mà ta sẽ trình bày sau.

```
FFT-BASE(a)
n ← length[a]                    ▷ n là một lũy thừa của 2.
for s ← 1 to lg n
    do m ← 2^s
       ω_m ← e^{2πi/m}
       for k ← 0 to n − 1 by m
           do ω ← 1
              for j ← 0 to m/2 − 1
                  do t ← ωA[k + j + m/2]
                     u ← A[k + j]
                     A[k + j] ← u + t
                     A[k + j + m/2] ← u − t
                     ω ← ω ω_m
```

Giờ đây ta trình bày phiên bản chung cuộc của mã FFT lặp lại của chúng ta, đảo hai vòng lặp bên trong để loại bỏ một phép tính chỉ số nào đó và sử dụng thủ tục phụ BIT-REVERSE-COPY(a, A) để chép vectơ a vào mảng A theo thứ tự ban đầu ở đó ta cần các giá trị.

```
ITERATIVE-FFT(a)
BIT-REVERSE-COPY(a, A)
n ← length[a]                    ▷ n là một lũy thừa của 2.
for s ← 1 to lg n
    do m ← 2^s
       ω_m ← e^{2πi/m}
       ω ← 1
       for j ← 0 to m/2 − 1
```

```
                do for k ← j to n − 1 by m
                    do t ← ω A[k + m/2]
                       u ← A[k]
                       A[k] ← u + t
                       A[k + m/2] ← u − t
              ω ← ω ω_m
return A
```

BIT-REVERSE-COPY đưa các thành phần của vectơ đầu vào a vào thứ tự mong muốn trong mảng A ra sao? Thứ tự ở đó các lá xuất hiện trong Hình 32.4 là "nhị phân nghịch đảo bit." Nghĩa là, nếu ta cho rev(k) là số nguyên lg n-bit được lập thành bằng cách đảo các bit của phần biểu diễn nhị phân của k, thì ta muốn đặt thành phần vectơ a_k trong vị trí mảng A[rev(k)]. Ví dụ, trong Hình 32.4, các lá xuất hiện theo thứ tự 0, 4, 2, 6, 1, 5, 3, 7; dãy này trong nhị phân là 000, 100, 010, 110, 001, 101, 011, 111, và trong nhị phân nghịch đảo bit ta có dãy 000, 001, 010, 011, 100, 101, 110, 111. Để thấy ta muốn thứ tự nhị phân nghịch đảo bit nói chung, ta lưu ý tại cấp trên cùng của cây, các chỉ số có bit cấp thấp là 0 được đặt trong cây con trái và các chỉ số có bit cấp thấp là 1 được đặt trong cây con phải. Lột bỏ bit cấp thấp tại mỗi cấp, ta tiếp tục tiến trình này xuống cây, cho đến khi ta có thứ tự nhị phân nghịch đảo bit tại các lá.

Bởi hàm rev(k) dễ tính toán, nên thủ tục BIT-REVERSE-COPY có thể được viết như sau.

```
BIT-REVERSE-COPY(a, A)
n ← length[a]
for k ← 0 to n − 1
     do A[rev(k)] ← a_k
```

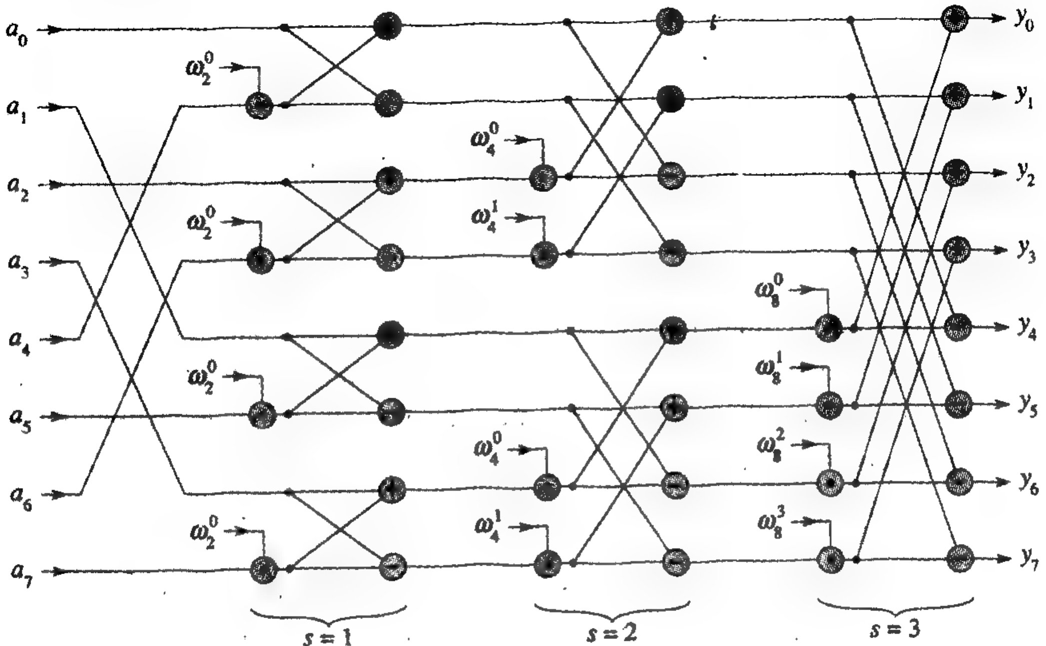
Kiểu thực thi FFT lặp lại chạy trong thời gian $\Theta(n \lg n)$. Lệnh gọi BIT-REVERSE-COPY(a, A) chắc chắn chạy trong $O(n \lg n)$ thời gian, bởi ta lặp lại n lần và có thể nghịch đảo một số nguyên giữa 0 và $n - 1$, với lg n bit, trong $O(\lg n)$ thời gian. (Trong thực tế, ta thường biết giá trị ban đầu của n trước, do đó ta ắt sẽ mã hóa một bảng ánh xạ k theo rev(k), khiến BIT-REVERSE-COPY chạy trong $\Theta(n)$ thời gian với một hằng ẩn thấp. Một cách khác, ta có thể dùng lược đồ bộ đếm nhị phân nghịch đảo khấu trừ thông minh hơn đã mô tả trong Bài toán 18-1.) Để hoàn thành phần chứng minh ITERATIVE-FFT chạy trong thời gian $\Theta(n \lg n)$, ta chứng tỏ $L(n)$, số lần thi hành thân của vòng lặp trong cùng (các dòng

9-12), là $\Theta(n \lg n)$. Ta có

$$
\begin{aligned}
L(n) &= \sum_{s=1}^{\lg n} \sum_{j=0}^{2^{s-1}-1} \frac{n}{2^s} \\
&= \sum_{s=1}^{\lg n} \frac{n}{2^s} \cdot 2^{s-1} \\
&= \sum_{s=1}^{\lg n} \frac{n}{2} \\
&= \Theta(n \lg n) .
\end{aligned}
$$

Một mạch FFT song song

Ta có thể khai thác nhiều tính chất đã cho phép ta thực thi một thuật toán FFT lặp lại hiệu quả để tạo ra một thuật toán song song hiệu quả cho FFT. (Xem Chương 29 để có phần mô tả của mô hình mạch tổ hợp.). Mạch tổ hợp PARALLEL-FFT tính toán FFT trên n đầu vào được nêu trong Hình 32.5 với $n = 8$. Mạch bắt đầu bằng một phép hoán vị nghịch đảo bit của các đầu vào, theo sau là $\lg n$ giai đoạn, mỗi giai đoạn bao gồm $n/2$ phép bướm được thi hành song song. Do đó bậc sâu của mạch là $\Theta(\lg n)$.



Hình 32.5 Một mạch tổ hợp PARALLEL-FFT tính toán FFT, ở đây được nêu trên $n = 8$ đầu vào. Các giai đoạn của các phép bướm được gán nhãn để tương ứng với các lần lặp lại của vòng lặp ngoài cùng của thủ tục FFT-BASE. Một FFT trên n đầu vào có thể được tính toán trong $\Theta(\lg n)$ độ sâu với $\Theta(n \lg n)$ thành phần tổ hợp.

Phần mút trái của mạch PARALLEL-FFT thực hiện phép hoán vị nghịch đảo bit, và phần còn lại bắt chước thủ tục FFT-BASE lặp lại. Ta vận dụng sự việc mỗi lần lặp lại của vòng lặp **for** ngoài cùng thực hiện $n/2$ phép toán bướm bậcc lập có thể được thực hiện song song. Giá trị của s trong mỗi lần lặp lại trong FFT-BASE tương ứng với một giai đoạn của các phép bướm nêu trong Hình 32.5. Bân trong giai đoạn s, với $s = 1, 2, \ldots, \lg n$, có $n/2^s$ nhóm các phép bướm (tương ứng với mỗi giá trị của k trong FFT-BASE), với 2^{s-1} phép bướm mỗi nhóm (tương ứng với mỗi giá trị của j trong FFT-BASE). Các phép bướm nêu trong Hình 32.5 tương ứng với các phép toán bướm của vòng lặp trong cùng (các dòng 8-11 của FFT-BASE). Cũng lưu ý các giá trị của ω được dùng trong các phép bướm tương ứng với các phép bướm được dùng trong FFT-BASE: trong giai đoạn s, ta dùng ω_m^m , ω_m^1 ,..., $\omega_m^{m/2-1}$, ở đó $m = 2^s$.

Bài tập

32.3-1

Nêu cách thức ITERATIVE-FFT tính toán DFT của vectơ đầu vào (0, 2, 3, −1, 4, 5, 7, 9).

32.3-2

Nêu cách thực thi một thuật toán FFT với phép hoán vị đảo bit xảy ra tại vào cuối, thay vì tại đầu, của phép tính. (*Mách nước:* Xét the DFT nghịch đảo.)

32.3-3

Để tính toán DFT_n, ta cần bao nhiêu thành phần phép cộng, phép trừ, và phép nhân, và bao nhiêu dây dẫn, trong mạch PARALLEL-FFT mô tả trong đoạn này? (Giả sử chỉ cần một dây dẫn để mang một số từ nơi này sang nơi khác.)

32.3-4 *

Giả sử rằng các bộ cộng trong mạch FFT đôi lúc thất bại theo cách chúng luôn tạo ra một kết xuất zero, độc lập với các đầu vào của chúng. Giả sử có chính xác một bộ cộng đã thất bại, nhưng bạn không biết bộ nào. Mô tả cách định danh bộ cộng bị thất bại bằng cách cung cấp các đầu vào cho mạch FFT chung và quan sát các kết xuất. Gắng tạo thủ tục của bạn hiệu quả.

Các Bài Toán

32-1 Phép nhân chia để trị

a. Nêu cách nhân hai đa thức tuyến tính $ax + b$ và $cx + d$ chỉ dùng ba phép nhân. (*Mách nước.* Một trong các phép nhân là $(a + b) \cdot (c + d)$.)

b. Nêu hai thuật toán chia để trị để nhân hai đa thức có cận bậc n chạy trong thời gian $\Theta(n^{\lg 3})$. Thuật toán đầu tiên sẽ chia các hệ số đa thức đầu vào thành một nửa cao và một nửa thấp, và thuật toán thứ hai sẽ chia chúng theo chỉ số của chúng là lẻ hoặc chẵn.

c. Chứng tỏ hai số nguyên n-bit có thể được nhân trong $O(n^{\lg 3})$ bước, ở đó mỗi bước hoạt bậcng trên tối đa một số lượng không đổi các giá trị 1-bit.

32-2 Các ma trận Toeplitz

Một ***ma trận Toeplitz*** là một ma trận $n \times n$ $A = (a_{ij})$ sao cho $a_{ij} = a_{i-1,j-1}$ với $i = 2, 3,\ldots, n$ và $j = 2, 3,\ldots, n$.

a. Tổng của hai ma trận Toeplitz có nhất thiết là Toeplitz không? Tích thì sao?

b. Mô tả cách biểu diễn một ma trận Toeplitz sao cho có thể cộng hai ma trận Toeplitz $n \times n$ trong $O(n)$ thời gian.

c. Nêu một thuật toán $O(n \lg n)$ thời gian để nhân một ma trận Toeplitz $n \times n$ với một vectơ có chiều dài n. Dùng phần biểu diễn của bạn từ phần (b).

d. Nêu một thuật toán hiệu quả để nhân hai ma trận Toeplitz $n \times n$. Phân tích thời gian thực hiện của nó.

32-3 Đánh giá tất cả các đạo hàm của một đa thức tại một điểm

Cho một đa thức $A(x)$ có cận bậc n, đạo hàm thứ t của nó được định nghĩa bởi

$$A^{(t)}(x) = \begin{cases} A(x) & \text{nếu } t = 0, \\ \frac{d}{dx} A^{(t-1)}(x) & \text{nếu } 1 \le t \le n-1, \\ 0 & \text{nếu } t \ge n. \end{cases}$$

Từ phần biểu diễn hệ số $(a_0, a_1,\ldots, a_{n-1})$ của $A(x)$ và một điểm đã cho x_0, ta muốn xác định $A^{(t)}(x_0)$ với $t = 0, 1,\ldots, n-1$.

a. Cho các hệ số $b_0, b_1,\ldots, b_{n-1}$ sao cho $n - 1$

$$A(x) = \sum_{j=0}^{n-1} b_j(x - x_0)^j ,$$

nêu cách tính toán $A^{(t)}(x_0)$, với $t = 0, 1, \ldots, n-1$, trong $O(n)$ thời gian.

b. Giải thích cách tìm $b_0, b_1, \ldots, b_{n-1}$ trong $O(n \lg n)$ thời gian, cho $A(x_0 + \omega_n^k)$ với $k = 0, 1, \ldots, n-1$.

c. Chứng minh rằng

$$A(x_0 + \omega_n^k) = \sum_{r=0}^{n-1} \left(\frac{\omega_n^{kr}}{r!} \sum_{j=0}^{n-1} f(j) g(r-j) \right) ,$$

ở đó $f(j) = a_j \bullet j!$ và

$$g(l) = \begin{cases} x_0^{-l}/(-l)! & \text{nếu } -(n-1) \le l \le 0 , \\ 0 & \text{nếu } 1 \le l \le (n-1) . \end{cases}$$

d. Giải thích cách đánh giá $A(x_0 + \omega_n^k)$ với $k = 0, 1, \ldots, n-1$ trong $O(n \lg n)$ thời gian. Kết luận tất cả các đạo hàm không hiển nhiên của $A(x)$ có thể được đánh giá tại x_0 trong $O(n \lg n)$ thời gian.

32-4 Đánh giá đa thức tại nhiều điểm

Ta đã quan sát thấy bài toán đánh giá một đa thức có cận bậc $n - 1$ tại một điểm đơn lẻ có thể giải quyết trong $O(n)$ thời gian bằng quy tắc Homer. Ta cũng đã khám phá rằng một đa thưc như vậy có thể được đánh giá tại tất cả n căn phức của đơn vị trong $O(n \lg n)$ thời gian bằng FFT. Giờ đây, ta sẽ nêu cách đánh giá một đa thức có cận bậc n tại n điểm tùy ý trong $O(n \lg^2 n)$ thời gian.

Để thực hiện, ta sẽ dùng sự việc cho rằng ta có thể tính toán phần còn lại của đa thức khi một đa thức như vậy được chia bởi một đa thức khác trong $O(n \lg n)$ thời gian, một kết quả mà ta mặc nhận mà không chứng minh. Ví dụ, phần còn lại của $3x^3 + x^2 - 3x + 1$ khi chia cho $x^2 + x + 2$ là

$$(3x^3 + x^2 - 3x + 1) \bmod (x^2 + x + 2) = 5x - 3 .$$

Cho phần biểu diễn hệ số của một đa thức $A(x) = \sum_{k=0}^{n-1} a_k x^k$ và n điểm $x_0, x_1, \ldots, x_{n-1}$, ta muốn tính toán n giá trị $A(x_0), A(x_1), \ldots, A(x_{n-1})$. Với $0 \le i \le j \le n-1$, định nghĩa các đa thức $P_{ij}(x) = \prod_{k=i}^{j} (x - x_k)$ và $Q_{ij}(x) = A(x) \bmod p_{ij}(x)$. Lưu ý, $Q_{ij}(x)$ có cận bậc tối đa $j - i$.

a. Chứng minh rằng $A(x) \bmod (x - z) = A(z)$ với bất kỳ điểm z.

b. Chứng minh rằng $Q_{kk}(x) = A(x_k)$ và rằng $Q_{0,n-1}(x) = A(x)$.

c. Chứng minh rằng với $i \le k \le j$, ta có $Q_{ik}(x) = Q_{ij}(x) \bmod P_{ik}(x)$ và $Q_{kj}(x) = Q_{ij}(x) \bmod P_{kj}(x)$.

d. Nêu một thuật toán $O(n \lg^2 n)$ thời gian để đánh giá $A(x_0), A(x_1), \ldots, A(x_{n-1})$.

32-5 FFT dùng số học modun

Như định nghĩa, Discrete Fourier Transform [biến đổi Fourier rời rạc] yêu cầu dùng các số phức, có thể dẫn đến sự mất chính xác vì các lỗi làm tròn. Với vài bài toán, đáp án được xem chỉ chứa các số nguyên, và có thể vận dụng một biến thức của FFT dựa trên số học modun để bảo đảm đáp án được tính toán chính xác. Một ví dụ về một bài toán như vậy đó là ví dụ về việc nhân hai đa thức với các hệ số số nguyên. Bài tập 32.2-6 cho ta một cách tiếp cận, dùng một modun có chiều dài $\Omega(n)$ bit để điều quản một DFT trên n điểm. Bài toán này cho một cách tiếp cận khác sử dụng một modun có chiều dài $O(\lg n)$ hợp lý hơn; nó yêu cầu bạn hiểu nội dung của Chương 33. Cho n là một lũy thừa của 2.

a. Giả sử rằng ta tìm kiếm k nhỏ nhất sao cho $p = kn + 1$ là số nguyên tố. Nêu một lập luận heuristic đơn giản cho biết tại sao ta có thể dự kiến k xấp xỉ bằng lg n. (Giá trị của k có thể lớn hơn hoặc nhỏ hơn nhiều, nhưng ta có thể có lý khi dự kiến xét $O(\lg n)$ giá trị ứng viên của k theo trung bình.) Chiều dài dự kiến của p so sánh như thế nào với chiều dài của n?

Cho g là một bộ sinh của $\mathbf{Z}^*_p$, và cho $w = g^k \bmod p$.

b. Chứng tỏ DFT và DFT nghịch đảo là các phép toán nghịch đảo được định nghĩa kỹ modulo p, ở đó w được dùng làm một căn thứ n chính của đơn vị.

c. Chứng tỏ FFT và nghịch đảo củanó có thể được thực hiện để làm việc modulo p trong thời gian $O(n \lg n)$, ở đó các phép toán trên các từ $O(\lg n)$ bit chiếm thời gian đơn vị. Giả sử thuật toán có p và w.

d. Tính toán DFT modulo $p = 17$ của vectơ (0, 5, 3, 7, 7, 2, 1, 6). Lưu ý, $g = 3$ là một bộ sinh của Z^*_{17}.

Ghi chú Chương

Press, Flannery, Teukolsky, và Vetterling [161, 162] có một phần một tốt về Fast Fourier Transform và các ứng dụng củanó. Để có phần giới thiệu tuyệt với về xử lý tín hiệu, một lĩnh vực ứng dụng FFT phổ dụng, bạn xem tài liệu của Oppenheim và Willsky [153].

Cooley và Tukey [51] được nhiều người công nhận là đã nghĩ ra FFT vào những năm 1960. FFT thực tế đã được khám phá nhiều lần trước đó, nhưng tầm quan trọng củanó chưa được nhận thức đầy đủ trước khi các máy tính số hóa hiện đại ra đời. Press, Flannery, Teukolsky, và Vetterling đã quy các nguồn gốc của phương pháp cho Runge và Konig (1924).

33 Các Thuật Toán Lý Thuyết Số

Đã có thời lý thuyết số được xem là một chủ đề tuy đẹp song phần lớn là vô dụng trong toán học thuần túy. Ngày nay các thuật toán lý thuyết số được dùng rộng rãi, một phần nhờ vào phát minh của các lược đồ mật mã hóa dựa trên các số nguyên tố lớn. Tính khả thi của các lược đồ này dựa vào khả năng của chúng ta có thể tìm ra các số nguyên tố lớn một cách dễ dàng, trong khi tính bảo mật của chúng dựa vào sự bất lực của chúng ta trong việc lấy thừa số tích của các số nguyên tố lớn. Chương này trình bày một ít về lý thuyết số và các thuật toán kết hợp làm cơ sở cho các ứng dụng như vậy.

Đoạn 33.1 giới thiệu các khái niệm cơ bản về lý thuyết số, như tính chia hết, tương đương modula, và phép thừa số hóa duy nhất. Đoạn 33.2 nghiên cứu một trong các thuật toán xưa nhất của thế giới: thuật toán Euclid để tính toán số chia chung lớn nhất của hai số nguyên. Đoạn 33.3 ôn lại các khái niệm về số học modula. Sau đó, Đoạn 33.4 nghiên cứu tập hợp các bội của một số đã cho a, modulo n, và nêu cách tìm tất cả các nghiệm cho phương trình $ax \equiv b \pmod{n}$ bằng cách dùng thuật toán Euclid. định lý phần dư Trung Quốc được trình bày trong Đoạn 33.5. Đoạn 33.6 xét các lũy thừa của một số đã cho a, modulo n, và trình bày một thuật toán bình phương lặp lại để tính toán hiệu quả a^b mod n, cho a, b, và n. Phép toán này là trọng tâm của việc thử tính nguyên hiệu quả. Sau đó, Đoạn 33.7 mô tả hệ mật mã khóa công RSA. Đoạn 33.8 mô tả một phép thử tính nguyên ngẫu nhiên hóa có thể được dùng để tìm các số nguyên tố lớn một cách hiệu quả, một công việc thiết yếu trong việc tạo các khóa cho hệ mật mã RSA. Cuối cùng, Đoạn 33.9 ôn lại một heuristic đơn giản nhưng hiệu quả để lấy thừa số các số nguyên nhỏ. Điều lạ đó là việc lấy thừa số là một bài toán mà người ta có thể muốn nó phải khó giải, bởi tính bảo mật của RSA tùy thuộc vào sự khó khăn của việc lấy thừa số các số nguyên lớn.

Kích cỡ của các đầu vào và mức hao phí của các phép tính số học

Bởi ta sẽ làm việc với các số nguyên lớn, nên ta cần điều chỉnh cách nghĩ về kích cỡ của một đầu vào và về mức hao phí của các phép toán số học cơ bản.

Trong chương này, một "đầu vào lớn" thường có nghĩa là một đầu vào chứa "các số nguyên lớn" thay vì một đầu vào chứa "nhiều số nguyên" (như để sắp xếp). Như vậy, ta sẽ đo kích cỡ của một đầu vào theo dạng *số lượng bit* cần thiết để biểu diễn đầu vào đó, chứ không chỉ là số lượng các số nguyên trong đầu vào. Một thuật toán có các đầu vào số nguyên $a_1, a_2, \ldots, a_k$ là một ***thuật toán thời gian đa thức*** nếu nó chạy trong thời gian đa thức trong $\lg a_1, \lg a_2, \ldots, \lg a_k$, nghĩa là, đa thức trong các chiều dài của các đầu vào đã mã hóa nhị phân của nó.

Trong phần lớn cuốn sách này, ta đã thấy tiện dụng khi xem các phép toán số học cơ bản (các phép nhân, chia, hoặc tính các số dư) như là các phép toán nguyên thủy chiếm một đơn vị thời gian. Bằng cách đếm số lượng các phép toán số học như vậy mà một thuật toán thực hiện, ta có một cơ sở để thực hiện một ước lượng hợp lý về thời gian thực hiện thực tế của thuật toán trên một máy tính. Tuy nhiên, các phép toán cơ bản có thể tốn thời gian, khi các đầu vào của chúng là lớn. Như vậy, để tiện dụng, người ta đo một thuật toán lý thuyết số yêu cầu bao nhiêu ***phép toán bit***. Trong mô hình này, một phép nhân hai số nguyên β-bit theo phương pháp bình thường sử dụng $\Theta(\beta^2)$ phép toán bit. Cũng vậy, phép toán chia một số nguyên β-bit cho một số nguyên ngắn hơn, hoặc phép toán lấy số dư của một số nguyên β-bit khi được chia với một số nguyên ngắn hơn, có thể được thực hiện trong thời gian $\Theta(\beta^3)$ bằng các thuật toán đơn giản. (Xem Bài tập 33.1-11.) Người ta cũng đã phát hiện các phương pháp nhanh hơn. Ví dụ, một phương pháp chia để trị đơn giản để nhân hai số nguyên β–bit có một thời gian thực hiện là $\Theta(\beta^{\lg_2 3})$, và phương pháp nhanh nhất được biết có một thời gian thực hiện là $\Theta(\beta \lg \beta \lg \lg \beta)$. Tuy nhiên, với các mục tiêu thực tiễn, thuật toán $\Theta(\beta^2)$ thường là tốt nhất, và ta sẽ dùng cận này làm cơ sở cho các phân tích của chúng ta.

Trong chương này, các thuật toán thường được phân tích theo dạng cả số lượng phép toán số học lẫn số lượng các phép toán bit mà chúng yêu cầu.

33.1 Các khái niệm lý thuyết số cơ bản

Đoạn này ôn lại khái quát lý thuyết số cơ bản liên quan đến tập hợp $\mathbf{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ của các số nguyên và tập hợp $\mathbf{N} = \{0, 1, 2, \ldots\}$ của các số tự nhiên.

Tính chia hết và các ước số

Khái niệm về một số nguyên chia được cho số nguyên khác là khái

niệm trung tâm trong lý thuyết về các con số. Hệ ký hiệu $d \mid a$ (đọc là "*d* ***chia hết*** *a*") có nghĩa là $a = kd$ với một số nguyên *k*. Mọi số nguyên đều chia hết 0. Nếu $a > 0$ và $d \mid a$, thì $|d| \leq |a|$. Nếu $d \mid a$, thì ta cũng nói rằng *a* là một ***bội*** của *d*. Nếu *d* không chia hết *a*, ta viết $d \nmid a$.

Nếu $d \mid a$ và $d \geq 0$, ta nói rằng *d* là một ***ước số*** của *a*. Lưu ý, $d \mid a$ nếu và chỉ nếu $-d \mid a$, sao cho tính tổng quát không bị mất khi định nghĩa các ước số là không âm, với sự hiểu biết rằng âm của một ước số bất kỳ của *a* cũng chia hết *a*. Một ước số của một số nguyên *a* ít nhất là 1 nhưng không lớn hơn $|a|$. Ví dụ, các ước số của 24 là 1, 2, 3, 4, 6, 8, 12, và 24.

Mọi số nguyên *a* đều chia được bởi ***các ước số hiển nhiên*** [trivial divisors] 1 và *a*. Các ước số không hiển nhiên của *a* còn được gọi là các ***thừa số*** của *a*. Ví dụ, các thừa số của 20 là 2, 4, 5, và 10.

Các số nguyên tố và hợp số

Một số nguyên $a > 1$ mà chỉ các ước số của nó là các ước số hiển nhiên 1 và *a* được xem là một ***số nguyên tố***. Các số nguyên tố có nhiều tính chất đặc biệt và đóng một vai trò quan trọng trong lý thuyết số. Theo thứ tự, các số nguyên tố nhỏ là 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59,....

Bài tập 33.1-1 yêu cầu bạn chứng minh có nhiều vô số các số nguyên tố. Một số nguyên $a > 1$ không phải là số nguyên tố được xem là một ***hợp số***. Ví dụ, 39 là hợp số bởi $3 \mid 39$. Số nguyên 1 được xem là một ***đơn vị*** và không phải là số nguyên tố cũng như hợp số. Cũng vậy, số nguyên 0 và tất cả các số nguyên âm đều không phải là số nguyên tố hoặc hợp số.

Định lý phép chia, các số dư, và tính tương đương modula

Cho một số nguyên *n*, các số nguyên có thể được phân hoạch thành các số là bội của *n* và các số không phải là bội của *n*. Phần lớn lý thuyết số đều dựa trên sự tinh chỉnh của phân hoạch này có được bằng cách phân loại các số không phải bội của *n* theo các số dư của chúng khi chia cho *n*. Định lý dưới đây là cơ sở cho sự tinh chỉnh này. Phần chứng minh của định lý này sẽ không được nêu ở đây (ví dụ, xem Niven và Zuckerman [151]).

Định lý 33.1 (Định lý phép chia)

Với bất kỳ số nguyên *a* và bất kỳ số nguyên dương *n*, ta có các số nguyên duy nhất *q* và *r* sao cho $0 \leq r < n$ và $a = qn + r$.

Giá trị $q = \lfloor a/n \rfloor$ là ***số thương*** của phép chia. Giá trị $r = a \bmod n$ là ***số dư*** (hoặc ***thặng dư***) của phép chia. Ta có $n \mid a$ nếu và chỉ nếu $a \bmod n =$

0. Vậy dẫn đến

$$a = \lfloor a/n \rfloor n + (a \bmod n) \qquad (33.1)$$

hoặc

$$a \bmod n = a - \lfloor a/n \rfloor n\,. \qquad (33.2)$$

Cho một khái niệm được định nghĩa kỹ của số dư của một số nguyên khi được chia với một số khác, để tiện dụng, ta cung cấp hệ ký hiệu đặc biệt để nêu rõ đẳng thức của các số dư. Nếu $(a \bmod n) = (b \bmod n)$, ta viết $a \equiv b \pmod n$ và nói rằng a là ***tương đương*** với b, modulo n. Nói cách khác, $a \equiv b \pmod n$ nếu a và b có cùng số dư khi được chia cho n. Theo tương đương, $a \equiv b \pmod n$ nếu và chỉ nếu $n \mid (b - a)$. Ta viết $a \not\equiv b \pmod n$ nếu a không tương đương với b, modulo n. Ví dụ, $61 \equiv 6 \pmod{11}$. Ngoài ra, $-13 \equiv 22 \equiv 2 \pmod 5$.

Các số nguyên có thể được chia thành n lớp tương đương theo các số dư của chúng modulo n. ***lớp tương đương modulo n*** chứa một số nguyên a là

$$[a]_n = \{a + kn : k \in \mathbf{Z}\}\,.$$

Ví dụ, $[3]_7 = \{..., -11, -4, 3, 10, 17, ...\}$; các cách biểu hiện khác cho tập hợp này là $[-4]_7$ và $[10]_7$. Viết $a \in [b]_m$ cũng giống như viết $a \equiv b \pmod n$. Tập hợp tất cả các lớp tương đương như vậy là

$$\mathbf{Z}_n = \{[a]_n : 0 \le a \le n - 1\}\,. \qquad (33.3)$$

Ta thường thấy phần định nghĩa

$$\mathbf{Z}_n = \{0, 1, ..., n - 1\}\,, \qquad (33.4)$$

sẽ được xem là tương đương với phương trình (33.3) với sự hiểu biết rằng 0 biểu diễn $[0]_n$, 1 biểu diễn $[1]_n$, và vân vân; mỗi lớp được biểu thị bởi thành phần không âm bé nhất của nó. Tuy nhiên, các lớp tương đương cơ sở phải được ghi nhớ. Ví dụ, một tham chiếu đến -1 dưới dạng một phần tử của $\mathbf{Z}_n$ là một tham chiếu đến $[n - 1]_n$, bởi $-1 \equiv n - 1 \pmod n$.

Các ước số chung và các ước số chung lớn nhất

Nếu d là một ước số của a và cũng là một ước số của b, thì d là một ***ước số chung*** của a và b. Ví dụ, các ước số của 30 là 1, 2, 3, 5, 6, 10, 15, và 30, và do đó các ước số chung của 24 và 30 là 1, 2, 3, và 6. Lưu ý, 1 là một ước số chung của bất kỳ hai số nguyên nào.

Một tính chất quan trọng của các ước số chung đó là

$$d \mid a \text{ và } d \mid b \text{ hàm ý } d \mid (a + b) \text{ và } d \mid (a - b)\,. \qquad (33.5)$$

Nói chung hơn, ta có

$d \mid a$ và $d \mid b$ hàm ý $d \mid (ax + by)$ (33.6)

với bất kỳ số nguyên x và y. Ngoài ra, nếu $a \mid b$, thì $|a| \le |b|$ hoặc $b = 0$, hàm ý rằng

$a \mid b$ và $b \mid a$ hàm ý $a = \pm b$. (33.7)

Ước số chung lớn nhất của hai số nguyên a và b, không phải cả hai zero, là số lớn nhất của các ước số chung của a và b; nó được ký hiệu là $\gcd(a, b)$. Ví dụ, $\gcd(24, 30) = 6$, $\gcd(5, 7) = 1$, và $\gcd(0, 9) = 9$. Nếu a và b không phải cả hai 0, thì $\gcd(a, b)$ là một số nguyên giữa 1 và $\min(|a|, |b|)$. Ta định nghĩa $\gcd(0, 0)$ là 0; định nghĩa này cần thiết để khiến các tính chất chuẩn của hàm gcd (như phương trình (33.11) dưới đây) trở thành hợp lệ toàn thể.

Dưới đây là các tính chất cơ bản của hàm gcd:

$$\gcd(a, b) = \gcd(b, a)\,, \quad (33.8)$$

$$\gcd(a, b) = \gcd(-a, b)\,, \quad (33.9)$$

$$\gcd(a, b) = \gcd(|a|, |b|)\,, \quad (33.10)$$

$$\gcd(a, 0) = |a|\,, \quad (33.11)$$

$$\gcd(a, ka) = |a| \text{ với bất kỳ } k \in \mathbf{Z}\,. \quad (33.12)$$

Định lý 33.2

Nếu a và b là số nguyên bất kỳ, không phải cả hai zero, thì $\gcd(a, b)$ là thành phần dương nhỏ nhất của tập hợp $\{ax + by : x, y \in \mathbf{Z}\}$ của các tổ hợp tuyến tính của a và b.

Chứng minh Cho s là tổ hợp tuyến tính dương nhỏ nhất như vậy của a và b, và cho $s = ax + by$ với một $x, y \in \mathbf{Z}$. Cho $q = \lfloor a/s \rfloor$. Như vậy, phương trình (33.2) hàm ý

$$\begin{aligned} a \bmod s &= a - qs \\ &= a - q(ax + by) \\ &= a(1 - qx) + b(-qy)\,, \end{aligned}$$

và như vậy $a \bmod s$ cũng là một tổ hợp tuyến tính của a và b. Nhưng, bởi $a \bmod s < s$, nên ta có $a \bmod s = 0$, bởi s là tổ hợp tuyến tính dương nhỏ nhất như vậy. Do đó, $s \mid a$ và, theo biện luận tương tự, $s \mid b$. Như vậy, s là một ước số chung của a và b, và do đó $\gcd(a, b) \ge s$. Phương trình (33.6) hàm ý rằng $\gcd(a, b) \mid s$, bởi $\gcd(a, b)$ chia hết cả a lẫn b và s là một tổ hợp tuyến tính của a và b. Nhưng $\gcd(a, b) \mid s$ và $s > 0$ hàm ý rằng $\gcd(a, b) \le s$. Việc tổ hợp $\gcd(a, b) \ge s$ và $\gcd(a, b) \le s$ cho ra $\gcd(a, b) = s$; ta kết luận rằng s là ước số chung lớn nhất của a và b.

Hệ luận 33.3

Với bất kỳ số nguyên a và b, nếu $d \mid a$ và $d \mid b$ thì $d \mid \gcd(a, b)$.

Chứng minh Hệ luận này là do phương trình (33.6), bởi $\gcd(a, b)$ là một tổ hợp tuyến tính của a và b theo Định lý 33.2.

Hệ luận 33.4

Với tất cả số nguyên a và b và bất kỳ số nguyên không âm n,

$\gcd(a, bn) = n \gcd(a, b)$.

Chứng minh Nếu $n = 0$, hệ luận là hiển nhiên. Nếu $n > 0$, thì $\gcd(an, bn)$ là thành phần dương nhỏ nhất của tập hợp $\{anx + bny\}$, là n nhân thành phần dương nhỏ nhất của tập hợp $\{ax + by\}$.

Hệ luận 33.5

Với tất cả các số nguyên dương n, a, và b, nếu $n \mid ab$ và $\gcd(a, n) = 1$, thì $n \mid b$.

Chứng minh Phần chứng minh được chừa làm Bài tập 33.1-4.

Các số nguyên tố cùng nhau

Hai số nguyên a, b được gọi là ***nguyên tố cùng nhau*** [relatively prime] nếu ước số chung duy nhất của chúng là 1, nghĩa là, nếu $\gcd(a, b) = 1$. Ví dụ, 8 và 15 là nguyên tố cùng nhau, bởi các ước số của 8 là 1, 2, 4, và 8, trong khi các ước số của 15 là 1, 3, 5, và 15. Định lý dưới đây phát biểu rằng nếu mỗi trong hai số nguyên là nguyên tố cùng nhau với một số nguyên p, thì tích của chúng là nguyên tố cùng nhau với p.

Định lý 33.6

Với bất kỳ số nguyên a, b, và p, nếu $\gcd(a, p) = 1$ và $\gcd(b, p) = 1$, thì $\gcd(ab, p) = 1$.

Chứng minh Nó là do Định lý 33.2 ở đó tồn tại các số nguyên x, y, x', và y' sao cho

$$ax + py = 1 ,$$

$$bx' + py' = 1 .$$

Nhân các phương trình này và dàn xếp lại, ta có

$$ab(xx') + p(ybx' + y'ax + pyy') = 1 .$$

Như vậy, do 1 là một tổ hợp tuyến tính dương của ab và p, ta chỉ cần dựa vào Định lý 33.2 để hoàn tất chứng minh.

Ta nói rằng các số nguyên n_1, n_2,..., n_k là ***nguyên tố cùng nhau theo từng cặp*** nếu, mỗi khi $i \neq j$, ta có $\gcd(n_i, n_j) = 1$.

Phép thừa số hóa duy nhất

Dưới đây là một sự việc tuy cơ bản song quan trọng về tính chia hết theo các số nguyên tố.

Định lý 33.7

Với tất cả các số nguyên tố p và tất cả số nguyên a, b, nếu $p \mid ab$, thì $p \mid a$ hoặc $p \mid b$.

Chứng minh Vì sự mâu thuẫn, mặc nhận $p \mid ab$ nhưng $p \nmid a$ và $p \nmid b$. Như vậy, $\gcd(a,p) = 1$ và $\gcd(b,p) = 1$, bởi các ước số duy nhất của p là 1 và p, và theo giả thiết p không chia hết a hoặc b. Như vậy, Định lý 33.6 hàm ý rằng $\gcd(ab, p) = 1$, mâu thuẫn với giả thiết của chúng ta rằng $p \mid ab$, bởi $p \mid ab$ hàm ý $\gcd(ab, p) = p$. Sự mâu thuẫn này hoàn tất phần chứng minh.

Một hệ quả của Định lý 33.7 đó là một số nguyên có một phép thừa số hóa duy nhất thành các số nguyên tố.

Định lý 33.8 (Phép thừa số hóa duy nhất)

Một hợp số nguyên a có thể được viết chính xác theo một cách dưới dạng một tích có dạng

$$a = p_1^{e1}, p_2^{e2} \ldots p_r^{er}$$

ở đó p_i là số nguyên tố, $p_1 < p_2 < \ldots < p_r$, và e_i là các số nguyên dương.

Chứng minh Phần chứng minh được để làm Bài tập 33.1-10.

Để lấy ví dụ, số 6000 có thể được lấy thừa số duy nhất dưới dạng $2^4 \bullet 3 \bullet 5^3$.

Bài tập

33.1-1

Chứng minh có nhiều vô số các số nguyên tố. (*Mách nước:* Chứng tỏ không có số nguyên tố nào $p_1, p_2, \ldots, p_k$ chia hết $(p_1p_2 \ldots p_k) + 1$.)

33.1-2

Chứng minh nếu $a \mid b$ và $b \mid c$, thì $a \mid c$.

33.1-3

Chứng minh nếu p là số nguyên tố và $0 < k < p$, thì $\gcd(k, p) = 1$.

33.1-4

Chứng minh Hệ luận 33.5.

33.1-5

Chứng minh nếu p là số nguyên tố và $0 < k < p$, thì $p \mid \binom{p}{k}$. Kết luận với tất cả số nguyên a, b, và các số nguyên tố p,

$(a + b)^p \equiv a^p + b^p \pmod p$.

33.1-6

Chứng minh nếu a và b là bất kỳ số nguyên sao cho $a \mid b$ và $b > 0$, thì

$(x \bmod b) \bmod a = x \bmod a$

với bất kỳ x. Dưới các giả thiết tương tự, chứng minh rằng

$x \equiv y \pmod b$ hàm ý $x \equiv y \pmod a$

với bất kỳ số nguyên x và y.

33.1-7

Với bất kỳ số nguyên $k > 0$, ta nói rằng một số nguyên n là một ***lũy thừa thứ k*** nếu ở đó tồn tại một số nguyên a sao cho $a^k = n$. Ta nói rằng $n > 1$ là một ***lũy thừa không hiển nhiên*** nếu nó là một lũy thừa thứ k với số nguyên $k > 1$. Nêu cách xác định nếu một số nguyên β-bit đã cho n là một lũy thừa không hiển nhiên trong thời gian đa thức trong β.

33.1-8

Chứng minh các phương trình (33.8)-(33.12).

33.1-9

Chứng tỏ toán tử gcd là kết hợp. Nghĩa là, chứng minh với tất cả các số nguyên a, b, và c,

$\gcd(a, \gcd(b, c)) = \gcd(\gcd(a, b), c)$.

33.1-10 *

Chứng minh Định lý 33.8.

33.1-11

Nêu các thuật toán hiệu quả cho các phép toán chia một số nguyên β-bit cho một số nguyên ngắn hơn và lấy số dư của một số nguyên β-bit khi được chia cho một số nguyên ngắn hơn. Các thuật toán của bạn sẽ chạy trong thời gian $O(\beta^2)$.

33.1-12

Nêu một thuật toán hiệu quả để chuyển đổi một số nguyên β-bit (nhị phân) đã cho thành một phần biểu diễn thập phân. Chứng tỏ nếu phép nhân hoặc phép chia của các số nguyên có chiều dài tối đa là β mất một

thời gian $M(\beta)$, thì phép chuyển đổi nhị phân thành thập phân có thể được thực hiện trong thời gian $\Theta(M(\beta)\lg \beta)$. (*Mách nước:* Dùng một cách tiếp cận chia để trị, có được các nửa đầu và cuối của kết quả với các lần đệ quy tách biệt.)

33.2 Ước số chung lớn nhất

Trong đoạn này, ta dùng thuật toán Euclid để tính toán ước số chung lớn nhất của hai số nguyên một cách hiệu quả. Sự phân tích thời gian thực hiện sẽ cho thấy một tuyến nối đáng ngạc nhiên với các số Fibonacci, cho ra một đầu vào trường hợp xấu nhất cho thuật toán Euclid.

Trong đoạn này, ta tự hạn chế vào các số nguyên không âm. Hạn chế này được biện minh bởi phương trình (33.10), phát biểu rằng $\gcd(a, b) = \gcd(|a|, |b|)$.

Theo nguyên tắc, ta có thể tính toán $\gcd(a, b)$ với các số nguyên dương a và b từ các phép thừa số hóa số nguyên tố của a và b. Quả vậy, nếu

$$a = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r} , \quad (33.13)$$

$$b = p_1^{f_1} p_2^{f_2} \cdots p_r^{f_r} , \quad (33.14)$$

với các số mũ zero đang được dùng để khiến tập hợp các số nguyên tố $p_1, p_2, \ldots, p_r$ là như nhau cho cả a lẫn b, thì

$$\gcd(a, b) = p_1^{\min(e_1,f_1)} p_2^{\min(e_2,f_2)} \cdots p_r^{\min(e_r,f_r)} . \quad (33.15)$$

Như sẽ nêu trong Đoạn 33.9, các thuật toán tốt nhất hiện nay để lấy thừa số không chạy trong thời gian đa thức. Như vậy, cách tiếp cận này để tính toán các ước số chung lớn nhất dường như không thể cho ra một thuật toán hiệu quả.

Thuật toán Euclid để tính toán các ước số chung lớn nhất dựa trên định lý dưới đây.

Định lý 33.9 (Định lý đệ quy GCD)

Với bất kỳ số nguyên không âm a và bất kỳ số nguyên dương b,

$\gcd(a, b) = \gcd(b, a \bmod b)$.

Chứng minh Ta sẽ chứng tỏ $\gcd(a, b)$ và $\gcd(b, a \bmod b)$ chia hết với nhau, sao cho theo phương trình (33.7) chúng phải bằng nhau (bởi cả hai đều không âm).

Trước tiên, ta chứng tỏ $\gcd(a, b) \mid \gcd(b, a \bmod b)$. Nếu ta cho $d = \gcd(a, b)$, thì $d \mid a$ và $d \mid b$. Theo phương trình (33.2), $(a \bmod b) = a - qb$,

ở đó $q = \lfloor a/b \rfloor$. Như vậy, do (a mod b) là một tổ hợp tuyến tính của a và b, nên phương trình (33.6) hàm ý rằng $d \mid (a \bmod b)$. Do đó, bởi $d \mid b$ và $d \mid (a \bmod b)$, nên Hệ luận 33.3 hàm ý rằng $d \mid \gcd(b, a \bmod b)$ hoặc, theo tương đương, rằng

$$\gcd(a, b) \mid \gcd(b, a \bmod b). \tag{33.16}$$

Chứng tỏ $\gcd(b, a \bmod b) \mid \gcd(a, b)$ hầu như cũng vậy. Nếu giờ đây ta cho $d = \gcd(b, a \bmod b)$, thì $d \mid b$ và $d \mid (a \bmod b)$. Bởi $a = qb + (a \bmod b)$, ở đó $q = \lfloor a/b \rfloor$, nên ta có a là một tổ hợp tuyến tính của b và (a mod b). Theo phương trình (33.6), ta kết luận $d \mid a$. Bởi $d \mid b$ và $d \mid a$, nên ta có $d \mid \gcd(a, b)$ theo Hệ luận 33.3 hoặc, theo tương đương, rằng

$$\gcd(b, a \bmod b) \mid \gcd(a, b). \tag{33.17}$$

Dùng phương trình (33.7) để tổ hợp các phương trình (33.16) và (33.17) sẽ hoàn tất phần chứng minh.

Thuật toán Euclid

Thuật toán gcd dưới đây được mô tả trong cuốn *Các thành phần* của Euclid (khoảng 300 B.C.), mặc dù thậm chí có thể có nguồn gốc sớm hơn. Nó được viết dưới dạng một chương trình đệ quy trực tiếp dựa vào Định lý 33.9. Các đầu vào a và b là các số nguyên không âm tùy ý.

```
EUCLID(a, b)
1 if b = 0
2    then return a
3    else return EUCLID(b, a mod b)
```

Để lấy ví dụ về cách thực hiện của EUCLID, ta xét phép tính của gcd(30, 21):

$$\begin{aligned} \text{EUCLID}(30, 21) &= \text{EUCLID}(21, 9) \\ &= \text{EUCLID}(9, 3) \\ &= \text{EUCLID}(3, 0) \\ &= 3\,. \end{aligned}$$

Trong phép tính này, ta có ba lần triệu gọi đệ quy của EUCLID.

Tính đúng đắn của EUCLID là do Định lý 33.9 và sự việc nếu thuật toán trả về a trong dòng 2, thì $b = 0$, do đó phương trình (33.11) hàm ý rằng $\gcd(a, b) = \gcd(a, 0) = a$. Thuật toán không thể đệ quy vô hạn, bởi đối số thứ hai giảm ngặt trong mỗi lệnh gọi đệ quy. Do đó, EUCLID luôn kết thúc bằng đáp án đúng.

Thời gian thực hiện của thuật toán Euclid

Ta phân tích thời gian thực hiện trường hợp xấu nhất của EUCLID dưới dạng một hàm có kích cỡ của a và b. Không để mất đi nhiều tính tổng quát, ta mặc nhận $a > b \geq 0$. Giả thiết này có thể được xác minh bằng nhận xét rằng nếu $b > a \geq 0$, thì EUCLID(a, b) lập tức thực hiện lệnh gọi đệ quy EUCLID(b, a). Nghĩa là, nếu đối số đầu tiên nhỏ hơn đối số thứ hai, EUCLID bỏ ra một lệnh gọi đệ quy tráo các đối số của nó rồi tiếp tục. Cũng vậy, nếu $b = a > 0$, thủ tục kết thúc sau một lệnh gọi đệ quy, bởi $a \bmod b = 0$.

Thời gian thực hiện chung của EUCLID tỷ lệ với số lượng lệnh gọi đệ quy mà nó thực hiện. Phân tích của chúng ta vận dụng các số Fibonacci F_k, được định nghĩa bởi phép truy toán (2.13).

Bổ đề 33.10

Nếu $a > b \geq 0$ và lần triệu gọi EUCLID(a, b) thực hiện $k \geq 1$ lệnh gọi đệ quy, thì $a \geq F_{k+2}$ và $b \geq F_{k+1}$.

Chứng minh Phần chứng minh dựa vào phương pháp quy nạp trên k. Với cơ sở của phương pháp quy nạp, cho $k = 1$. Như vậy, $b \geq 1 = F_2$, và bởi $a > b$, nên ta phải có $a \geq 2 = F_3$. Bởi $b > (a \bmod b)$, nên trong mỗi lệnh gọi đệ quy đối số đầu tiên hoàn toàn lớn hơn đối số thứ hai; do đó giả thiết $a > b$ áp dụng cho mỗi lệnh gọi đệ quy.

Theo quy nạp, ta mặc nhận rằng bổ đề là đúng nếu $k - 1$ lệnh gọi đệ quy được thực hiện; như vậy ta sẽ chứng minh nó là đúng với k lệnh gọi đệ quy. Bởi $k > 0$, ta có $b > 0$, và EUCLID(a, b) gọi EUCLID(b, $a \bmod b$) theo đệ quy, mà đến lượt nó sẽ thực hiện k - 1 lệnh gọi đệ quy. Như vậy, giả thuyết quy nạp hàm ý rằng $b \geq F_{k+1}$ (do đó là phần chứng minh của bổ đề), và $(a \bmod b) \geq F_k$. Ta có

$$\begin{aligned} b + (a \bmod b) &= b + (a - \lfloor a/b \rfloor b) \\ &\leq a, \end{aligned}$$

bởi $a > b > 0$ hàm ý $\lfloor a/b \rfloor \geq 1$. Như vậy,

$$\begin{aligned} a &\geq b + (a \bmod b) \\ &\geq F_{k+1} + F_k \\ &= F_{k+2}. \end{aligned}$$

Định lý dưới đây là hệ luận tức thời của bổ đề này.

Định lý 33.11 (Định lý Lamé)

Với bất kỳ số nguyên $k \geq 1$, nếu $a > b \geq 0$ và $b < F_{k+1}$, thì lần triệu gọi EUCLID(a, b) sẽ tạo ít k lệnh gọi đệ quy hơn.

Ta có thể chứng tỏ cận trên của Định lý 33.11 là khả dĩ tốt nhất. Các số Fibonacci liên tục là một đầu vào trường hợp xấu nhất cho EUCLID. Bởi EUCLID(F_3, F_2) thực hiện chính xác một lệnh gọi đệ quy, và bởi với $k \geq 2$ ta có $F_{k+1} \bmod F_k = F_{k-1}$, ta cũng có

$$\begin{aligned} \gcd(F_{k+1}, F_k) &= \gcd(F_k, (F_{k+1} \bmod F_k)) \\ &= \gcd(F_k, F_{k-1}) . \end{aligned}$$

Như vậy, EUCLID(F_{k+1}, F_k) đệ quy chính xác $k - 1$ lần, thỏa cận trên của Định lý 33.11.

Bởi F_k xấp xỉ bằng $\phi^k/\sqrt{5}$, ở đó ϕ là tỷ số vàng $(1 + \sqrt{5})/2$ được định nghĩa bởi phương trình (2.14), nên số lần lệnh gọi đệ quy trong EUCLID là $O(\lg b)$. (Xem Bài tập 33.2-5 để có một cận chặt hơn.) Vậy dẫn đến nếu EUCLID được áp dụng cho hai con số β-bit, thì nó sẽ thực hiện $O(\beta)$ phép toán số học và $O(\beta^3)$ bit các phép toán (giả định phép nhân và phép chia of β-bit các con số take $O(\beta^2)$ phép toán bit). Bài toán 33-2 yêu cầu bạn nêu một cận $O(\beta^2)$ trên số lượng các phép toán bit.

Dạng mở rộng của thuật toán Euclid

Giờ đây, ta viết lại thuật toán Euclid để tính toán thông tin hữu ích bổ sung. Cụ thể, ta mở rộng thuật toán để tính toán các hệ số số nguyên x và y sao cho

$$d = \gcd(a, b) = ax + by . \qquad (33.18)$$

Lưu ý, x và y có thể là zero hoặc âm. Ta sẽ tìm các hệ số hữu ích này về sau để tính các nghịch đảo nhân modula. Thủ tục EXTENDED-EUCLID sử dụng một cặp số nguyên tùy ý làm đầu vào và trả về một bộ ba có dạng (d, x, y) thỏa phương trình (33.18).

a	b	$\lfloor a/b \rfloor$	d	x	y
99	78	1	3	−11	14
78	21	3	3	3	−11
21	15	1	3	−2	3
15	6	2	3	1	−2
6	3	2	3	0	1
3	0	–	3	1	0

Hình 33.1 Một ví dụ về phép toán của EXTENDED-EUCLID trên các đầu vào 99 và 78. Mỗi dòng nêu cho một cấp độ quy: các giá trị của các đầu vào a và b, giá trị đã tính toán $\lfloor a/b \rfloor$ và các giá trị d, x, và y được trả về. Bộ ba (d, x, y) được trả về trở thành bộ ba (d', x', y') được dùng trong phép tính tại cấp độ quy cao hơn. Lệnh gọi EXTENDED-EUCLID(99, 78) trả về (3, −11, 14), do đó gcd(99, 78) = 3 và gcd(99, 78) = 3 = 99 • (−11) + 78 • 14.

```
EXTENDED-EUCLID(a, b)
1 if b = 0
2     then return (a, 1, 0)
3 (d', x', y') ← EXTENDED-EUCLID(b, a mod b)
4 (d, x, y) ← (d', y', x' − ⌊a / b⌋y')
5 return (d, x, y)
```

Hình 33.1 minh họa việc thi hành EXTENDED-EUCLID với phép tính của gcd(99, 78).

Thủ tục EXTENDED-EUCLID là một biến thể của thủ tục EUCLID. Dòng 1 tương đương với trắc nghiệm "$b = 0$" trong dòng 1 của EUCLID. Nếu $b = 0$, thì EXTENDED-EUCLID trả về không những $d = a$ trong dòng 2, mà còn các hệ số $x = 1$ và $y = 0$, sao cho $a = ax + by$. Nếu $b \neq 0$, trước tiên EXTENDED-EUCLID tính toán (d', x', y') sao cho $d' = \gcd(b, a \bmod b)$ và

$$d' = bx' + (a \bmod b)y' . \qquad (33.19)$$

Cũng như với EUCLID, trong trường hợp này ta có $d = \gcd(a, b) = d' = \gcd(b, a \bmod b)$. Để được x và y sao cho $d = ax + by$, ta bắt đầu bằng cách viết lại phương trình (33.19) dùng phương trình $d = d'$ và phương trình (33.2):

$$\begin{aligned} d &= bx' + (a - \lfloor a/b \rfloor b)y' \\ &= ay' + b\,(x' \lfloor a/b \rfloor y') . \end{aligned}$$

Như vậy, chọn $x = y'$ và $y = x' - \lfloor a/b \rfloor y'$ thỏa phương trình $d = ax + by$, chứng minh tính đúng đắn EXTENDED-EUCLID.

Bởi số lượng lệnh gọi đệ quy thực hiện trong EUCLID bằng với số lượng lệnh gọi đệ quy thực hiện trong EXTENDED-EUCLID, nên các thời gian thực hiện của EUCLID và EXTENDED-EUCLID là như nhau, theo một thừa số bất biến. Nghĩa là, với $a > b > 0$, số lượng lệnh gọi đệ quy là $O(\lg b)$.

Bài tập

33.2-1

Chứng minh các phương trình (33.13)-(33.14) hàm ý phương trình (33.15).

33.2-2

Tính toán các giá trị (d, x, y) là kết xuất bởi lần triệu gọi EXTENDED-EUCLID(899, 493).

33.2-3

Chứng minh với tất cả các số nguyên a, k, và n,

$\gcd(a, n) = \gcd(a + kn, n)$.

33.2-4

Viết lại EUCLID theo một dạng lặp lại chỉ sử một lượng bộ nhớ không đổi (nghĩa là, chỉ lưu trữ một lượng bất biến các giá trị số nguyên).

33.2-5

Nếu $a > b \geq 0$, chứng tỏ lần gọi EUCLID(a, b) thực hiện tối đa $1 + \log_\phi b$ lệnh gọi đệ quy. Cải thiện cận này thành $1 + \log_\phi (b/\gcd(a, b))$.

33.2-6

EXTENDED-EUCLID(F_{k+1}, F_k) trả về nội dung gì? Chứng minh đáp án của bạn là đúng.

33.2-7

Xác minh kết xuất (d, x, y) của EXTENDED-EUCLID(a, b) bằng cách chứng tỏ nếu $d \mid a$, $d \mid b$, và $d = ax + by$, thì $d = \gcd(a, b)$.

33.2-8

Định nghĩa hàm gcd cho trên hai đối số bằng phương trình đệ quy $\gcd(a_0, a_1, \ldots, n) = \gcd(a_0, \gcd(a_1, \ldots, a_n))$. Chứng tỏ gcd trả về cùng đáp án độc lập với thứ tự ở đó các đối số của nó được chỉ định. Nêu cách tìm $x_0, x_1, \ldots, x_n$ sao cho $\gcd(a_0, a_1, \ldots, a_n) = a_0x_0 + a_1x_1 + \ldots + a_nx_n$. Chứng tỏ số lượng phép chia mà thuật toán của bạn thực hiện là $O(n + \lg(\max_i a_i))$.

33.2-9

Định nghĩa $\text{lcm}(a_1, a_2, \ldots, a_n)$ là ***bội chung bé nhất*** của các số nguyên $a_1, a_2, \ldots, a_n$ nghĩa là, số nguyên không âm bé nhất là một bội của từng a_i. Nêu cách tính toán $\text{lcm}(a_1, a_2, \ldots, a_n)$ một cách hiệu quả dùng phép toán gcd (hai đối số) làm một chương trình con.

33.2-10

Chứng minh n_1, n_2, n_3, và n_4 là nguyên tố cùng nhau theo từng cặp nếu và chỉ nếu $\gcd(n_1n_2, n_3n_4) = \gcd(n_1n_3, n_2n_4) = 1$. Một cách chung hơn, chứng tỏ rằng $n_1, n_2, \ldots, n_k$ là nguyên tố cùng nhau theo từng cặp nếu và chỉ nếu một tập hợp của $\lceil \lg k \rceil$ cặp các con số phái sinh từ n_i là nguyên tố cùng nhau.

33.3 Số học modula

Một cách không chính thức, ta có thể cho rằng số học modula dạng số học như bình thường trên các số nguyên, ngoại trừ nếu ta đang làm việc modulo n, thì mọi kết quả x được thay bằng thành phần của $\{0, 1, \ldots, n-1\}$ tương đương với x, modulo n (nghĩa là, x được thay bằng $x \bmod n$). Mô hình không chính thức này là đủ nếu ta bám vào các phép toán cộng, trừ, và nhân. Một mô hình chính thức hơn cho số học modula, mà ta đưa ra ở đây, được mô tả tốt nhất trong khuôn khổ lý thuyết nhóm.

Các nhóm hữu hạn

Một ***nhóm*** $(S, \oplus)$ là một tập hợp S chung nhau với một phép toán nhị phân $\oplus$ được định nghĩa trên S có các tính chất dưới đây.

1. **Đóng [closure]:** Với tất cả $a, b \in S$, ta có $a \oplus b \in S$.

2. **Đồng nhất thức [identity]:** Có một thành phần $e \in S$ sao cho $e \oplus a = a \oplus e = a$ với tất cả $a \in S$.

3. **Tính kết hợp [associativity]:** Với tất cả $a, b, c \in S$, ta có $(a \oplus b) \oplus c = a \oplus (b \oplus c)$.

4. **Nghịch đảo [inverses]:** Với mỗi $a \in S$, ở đó tồn tại một thành phần duy nhất $b \in S$ sao cho $a \oplus b = b \oplus a = e$.

Để lấy ví dụ, xét nhóm quen thuộc $(\mathbf{Z}, +)$ gồm các số nguyên $\mathbf{Z}$ dưới phép toán cộng: 0 là đồng nhất thức, và nghịch đảo của a là $-a$. Nếu một nhóm $(S, \oplus)$ thỏa *định luật giao hoán* $a \oplus b = b \oplus a$ với tất cả $a, b \in S$, thì nó là một ***nhóm abel***. Nếu một nhóm $(S, \oplus)$ thỏa $|S| < \infty$, thì nó là một ***nhóm hữu hạn***.

Các nhóm được định nghĩa bằng phép cộng và phép nhân modula

Ta có thể lập thành hai nhóm *abel* hữu hạn bằng cách dùng phép cộng và phép nhân modulo n, ở đó n là một số nguyên dương. Các nhóm này dựa trên các lớp tương đương của các số nguyên modulo n, được định nghĩa trong Đoạn 33.1.

Để định nghĩa một nhóm trên $\mathbf{Z}_n$, ta cần có các phép toán nhị phân thích hợp, mà ta có được bằng cách định nghĩa lại các phép toán bình thường của phép cộng và phép nhân. Ta có thể dễ dàng định nghĩa các phép toán cộng và nhân cho $\mathbf{Z}_n$, bởi lớp tương đương của hai số nguyên xác định duy nhất lớp tương đương của tổng hoặc tích của chúng. Nghĩa là, nếu $a \equiv a' \pmod n$ và $b \equiv b' \pmod n$, thì

$$a + b \equiv a' + b' \pmod n,$$

$$ab \equiv a'b' \pmod n.$$

$+_6$	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	2	3	4	5	0
2	2	3	4	5	0	1
3	3	4	5	0	1	2
4	4	5	0	1	2	3
5	5	0	1	2	3	4

(a)

$\cdot_{15}$	1	2	4	7	8	11	13	14
1	1	2	4	7	8	11	13	14
2	2	4	8	14	1	7	11	13
4	4	8	1	13	2	14	7	11
7	7	14	13	4	11	2	1	8
8	8	1	2	11	4	13	14	7
11	11	7	14	2	13	1	8	4
13	13	11	7	1	14	8	4	2
14	14	13	11	8	7	4	2	1

(b)

Hình 33.2 Hai nhóm hữu hạn. Các lớp tương đương được ký hiệu bằng các thành phần biểu diễn của chúng. **(a)** Nhóm $(\mathbf{Z}_6, +_6)$. **(b)** Nhóm $(\mathbf{Z}^*_{15}, \cdot_{15})$.

Như vậy, ta định nghĩa phép cộng và phép nhân modulo n, được thể hiện $+_n$ và $\cdot_n$ như sau:

$$[a]_n +_n [b]_n = [a + b]_n ,$$

$$[a]_n \cdot_n [b]_n = [ab]_n .$$

(Phép trừ có thể được định nghĩa tương tự trên $\mathbf{Z}_n$ by $[a]_n -_n [b]_n = [a - b]_n$, nhưng phép chia thì phức tạp hơn, như sẽ thấy.) Các sự việc này xác minh thói quen tiện dụng và chung trong việc dùng thành phần không âm bé nhất của mỗi lớp tương đương làm phần biểu diễn của nó khi thực hiện các phép tính trong $\mathbf{Z}_n$. Phép cộng, phép trừ, và phép nhân được thực hiện như thường lệ trên các đại diện, nhưng mỗi kết quả x được thay bằng đại biểu của lớp của nó (nghĩa là, theo $x \bmod n$).

Dùng định nghĩa này của phép cộng modulo n, ta định nghĩa ***nhóm cộng tính modulo n*** dưới dạng $(\mathbf{Z}_n, +_n)$. Kích cỡ này của nhóm cộng tính modulo n là $|\mathbf{Z}_n| = n$. Hình 33.2(a) cho ta bảng phép toán của nhóm $(\mathbf{Z}_6, +_6)$.

Định lý 33.12

Hệ thống $(\mathbf{Z}_n, +_n)$ là một nhóm abel hữu hạn.

Chứng minh Tính kết hợp và tính giao hoán của $+_n$ là do tính kết hợp và tính giao hoán của +:

$$\begin{aligned}([a]_n +_n [b]_n) +_n [c]_n &= [(a + b) + c]_n \\ &= [a + (b + c)]_n \\ &= [a]_n +_n ([b]_n +_n [c]_n) ,\end{aligned}$$

$$[a]_n +_n [b]_n = [a+b]_n = [b+a]_n = [b]_n +_n [a]_n .$$

Thành phần đồng nhất thức của $(\mathbf{Z}_n, +_n)$ là 0 (nghĩa là, $[0]_n$). Nghịch đảo (cộng tính) của một thành phần a (nghĩa là, $[a]_n$) là thành phần $-a$ (nghĩa là, $[-a]_n$ hoặc $[n-a]_n$), bởi $[a]_n +_n [-a]_n = [a-a]_n = [0]_n$.

Dùng phần định nghĩa của phép nhân modulo n, ta định nghĩa *nhóm nhân modulo n* dưới dạng $(\mathbf{Z}^*_n, \cdot_n)$. Các thành phần của nhóm này là tập hợp $\mathbf{Z}^*_n$ gồm các thành phần trong $\mathbf{Z}_n$ là nguyên tố cùng nhau đối với n:

$$\mathbf{Z}^*_n = \{[a]_n \in \mathbf{Z}_n : \gcd(a, n) = 1\} .$$

Để xem $\mathbf{Z}^*_n$ được định nghĩa kỹ, lưu ý với $0 \le a < n$, ta có $a \equiv (a + kn) \pmod n$ với tất cả các số nguyên k. Do đó, theo Bài tập 33.2-3, $\gcd(a, n) = 1$ hàm ý $\gcd(a + kn, n) = 1$ với tất cả các số nguyên k. Bởi $[a]_n = \{a + kn : k \in \mathbf{Z}\}$, tập hợp $\mathbf{Z}^*_n$ được định nghĩa kỹ. Một ví dụ về một nhóm như vậy là

$$\mathbf{Z}^*_{15} = \{1,2,4,7,8,11,13,14\} ,$$

ở đó phép toán nhóm là phép nhân modulo 15. (Ở đây ta thể hiện một thành phần $[a]_{15}$ dưới dạng a.) Hình 33.2(b) nêu nhóm $(\mathbf{Z}^*_{15}, \cdot_{15})$. Ví dụ, $8 \cdot 11 \equiv 13 \pmod{15}$, làm việc trong $\mathbf{Z}^*_{15}$. Đồng nhất thức của nhóm này là 1.

Định lý 33.13

Hệ thống $(\mathbf{Z}^*_n, \cdot_n)$ là một nhóm hữu hạn abel.

Chứng minh Định lý 33.6 hàm ý $(\mathbf{Z}^*_n, \cdot_n)$ là đóng. Tính kết hợp và tính giao hoán có thể được chứng minh với $\cdot_n$ như trong trường hợp với $+_n$ trong phần chứng minh của Định lý 33.12. Thành phần đồng nhất thức là $[1]_n$. Để nêu sự tồn tại của các nghịch đảo, cho a là một thành phần của $\mathbf{Z}^*_n$ và cho (d, x, y) là kết xuất của EXTENDED-EUCLID(a, n). Như vậy $d = 1$, bởi $a \in \mathbf{Z}^*_n$, và

$$ax + ny = 1$$

hoặc, theo tương đương,

$$ax \equiv 1 \pmod n .$$

Như vậy, $[x]_n$ là một nghịch đảo nhân của $[a]_n$, modulo n. Phần chứng minh rằng các nghịch đảo được định nghĩa duy nhất sẽ được hoãn lại cho đến Hệ luận 33.26.

Khi làm việc với các nhóm $(\mathbf{Z}_n, +_n)$ và $(\mathbf{Z}_n, \cdot_n)$ trong phần còn lại của

chương này, ta theo thói quen tiện dụng trong việc thể hiện các lớp tương đương bằng các thành phần biểu diễn của chúng và thể hiện các phép toán $+_n$ và $\cdot_n$ bằng các hệ ký hiệu số học bình thường + và . (hoặc đặt kề nhau) theo thứ tự nêu trên. Ngoài ra, các tương đương modulo n cũng có thể được diễn dịch dưới dạng các phương trình trong $\mathbf{Z}_n$. Ví dụ, hai phát biểu dưới đây là tương đương:

$$ax \equiv b \pmod n,$$

$$[a]_n \cdot_n [x]_n = [b]_n .$$

Để tiện dụng hơn, đôi lúc ta tham chiếu một nhóm $(S, \oplus)$ đơn thuần dưới dạng S khi phép toán được hiểu từ ngữ cảnh. Như vậy, ta có thể tham chiếu các nhóm $(\mathbf{Z}_n, +_n)$ và $(\mathbf{Z}^*_n, \cdot_n)$ *as* $\mathbf{Z}_n$ và $\mathbf{Z}^*_n$, theo thứ tự nêu trên.

Nghịch đảo (nhân) của một thành phần a được ký hiệu là $(a^{-1} \bmod n)$. Phép nhân trong $\mathbf{Z}^*_n$ được định nghĩa bởi phương trình $a / b \equiv ab^{-1} \pmod n$. Ví dụ, trong $\mathbf{Z}^*_{15}$ ta có $7^{-1} \equiv 13 \pmod{15}$, bởi $7 \cdot 13 \equiv 91 \equiv 1 \pmod{15}$, sao cho $4 / 7 \equiv 4 \cdot 13 \equiv 7 \pmod{15}$.

Kích cỡ của $\mathbf{Z}^*_n$ được ký hiệu $\phi(n)$. Hàm này, có tên là ***hàm phi của Euler***, thỏa phương trình

$$\phi(n) = n \prod_{p | n} \left(1 - \frac{1}{p}\right) \qquad (33.20)$$

ở đó p chạy trên tất cả các số nguyên tố chia n (kể cả chính n, nếu n là số nguyên tố). Ta sẽ không chứng minh công thức này ở đây. Theo trực giác, ta bắt đầu bằng một danh sách n số dư $\{0, 1, \ldots, n-1\}$ rồi, với mỗi số nguyên tố p chia hết n, xóa đi mọi bội của p trong danh sách. Ví dụ, bởi các ước số nguyên tố của 45 là 3 và 5,

$$\begin{aligned} \phi(45) &= 45\left(1 - \frac{1}{3}\right)\left(1 - \frac{1}{5}\right) \\ &= 45\left(\frac{2}{3}\right)\left(\frac{4}{5}\right) \\ &= 24 . \end{aligned}$$

Nếu p là số nguyên tố, thì $\mathbf{Z}^*_p = \{1, 2, \ldots, p-1\}$, và

$$\phi p = p - 1 . \qquad (33.21)$$

Nếu n là hợp số, thì $\phi(n) < n - 1$.

Các nhóm con

Nếu $(S, \oplus)$ là một nhóm, $S' \subseteq S$, và $(S', \oplus)$ cũng là một nhóm, thì $(S', \oplus)$ được gọi là một ***nhóm con*** của $(S, \oplus)$. Ví dụ, thậm chí các số

nguyên hình thành một nhóm con các số nguyên dưới phép toán cộng. Định lý dưới đây cung cấp một công cụ hữu ích để nhận ra các nhóm con.

Định lý 33.14 (Một tập hợp con đóng của một nhóm hữu hạn là một nhóm con) Nếu $(S, \oplus)$ là một nhóm hữu hạn và S' là bất kỳ tập hợp con của S sao cho $a \oplus b \in S'$ với tất cả $a, b \in S'$, thì $(S', \oplus)$ là một nhóm con của $(S, \oplus)$.

Chứng minh Phần chứng minh được dành làm Bài tập 33.3-2.

Ví dụ, tập hợp {0, 2, 4, 6} hình thành một nhóm con của $\mathbf{Z}_8$, bởi nó được đóng dưới phép toán + (nghĩa là, nó được đóng dưới $+_8$).

Định lý dưới đây cung cấp một sự ràng buộc cực kỳ hữu ích trên kích cỡ của một nhóm con.

Định lý 33.15 (Định lý Lagrange)

Nếu $(S, \oplus)$ là một nhóm hữu hạn và $(S', \oplus)$ là một nhóm con của $(S, \oplus)$, thì $|S'|$ là một ước số của $|S|$.

Một nhóm con S' của một nhóm S được gọi là một nhóm con ***riêng*** [proper subgroup] nếu $S' \neq S$. Hệ luận dưới đây được dùng trong phân tích của chúng ta về thủ tục phép thử tính nguyên Miller-Rabin trong Đoạn 33.8.

Hệ luận 33.16

Nếu S' là một nhóm con riêng của một nhóm hữu hạn S, thì $|S'| \leq |S|/2$.

Các nhóm con được phát sinh bởi một thành phần

Định lý 33.14 cung cấp một cách thú vị để tạo một nhóm con của một nhóm hữu hạn $(S, \oplus)$: chọn một thành phần a và lấy tất cả các thành phần có thể được phát sinh từ a dùng phép toán nhóm. Cụ thể, định nghĩa $a^{(k)}$ với $k \geq 1$ bằng

$$a^{(k)} = \bigoplus_{i=1}^{k} a = \underbrace{a \oplus a \oplus \ldots \oplus a}_{k} .$$

Ví dụ, nếu ta lấy $a = 2$ trong nhóm $\mathbf{Z}_6$, dãy $a^{(1)}, a^{(2)}, \ldots$ là

2, 4, 0, 2, 4, 0, 2, 4, 0,....

Trong nhóm $\mathbf{Z}_n$, ta có $a^{(k)} = ka \bmod n$, và trong nhóm $\mathbf{Z}^*_n$, ta có $a^{(k)} = a^k \bmod n$. ***Nhóm con được phát sinh bởi a,*** được ký hiệu $\langle a \rangle$ hoặc $(\langle a \rangle, \oplus)$, được định nghĩa bởi

$$\langle a \rangle = \{a(k) : k \geq 1\} .$$

Ta nói rằng a ***phát sinh*** nhóm con $<a>$ hoặc a là một ***bộ sinh*** của $<a>$.

Bởi S là hữu hạn, nên $\langle a\rangle$ là một tập hợp con hữu hạn của S, có thể bao gồm toàn bộ S. Bởi tính kết hợp của ED hàm ý

$a^{(i)} \oplus a^{(j)} = a^{(i+j)}$,

$<a>$ là đóng và do đó, theo Định lý 33.14, $\langle a\rangle$ là một nhóm con của S. Ví dụ, trong $\mathbf{Z}_6$, ta có

$<0> \;=\; \{0\}$

$<1> \;=\; \{0, 1, 2, 3, 4, 5\}$,

$<2> \;=\; \{0, 2, 4\}$.

Cũng vậy, trong $\mathbf{Z}^*_7$, ta có

$<1> \;=\; \{1\}$,

$<2> \;=\; \{1, 2, 4\}$,

$<3> \;=\; \{1, 2, 3, 4, 5, 6\}$.

Thứ tự của a (trong nhóm S), được ký hiệu là ord(a), được định nghĩa là $t > 0$ bé nhất sao cho $a^{(t)} = e$.

Định lý 33.17

Với bất kỳ nhóm hữu hạn $(S, \oplus)$ và bất kỳ $a \in S$, thứ tự của một thành phần bằng với kích cỡ của nhóm con mà nó phát sinh, hoặc ord(a) $= |<a>|$.

Chứng minh Cho t = ord(a). Bởi $a^{(t)} = e$ và $a^{(t+k)} = a^{(t)} \oplus a^{(k)} = a^{(k)}$ với $k \geq 1$, nếu $i > t$, thì $a^{(i)} = a^{(j)}$ với một $j < i$. Như vậy, không gặp thành phần nào mới sau $a^{(t)}$, và $<a> = \{a^{(1)}, a^{(2)},..., a^{(t)}\}$. Để chứng tỏ $|<a>| = t$, vì sự mâu thuẫn ta giả sử rằng $a^{(i)} = a^{(j)}$ với một i, j thỏa $1 \leq i < j \leq t$. Như vậy, $a^{(i+k)} = a^{(j+k)}$ với $k \geq 0$. Nhưng điều này hàm ý rằng $a^{(i+(t-j))} = a^{(j+(t-j))} = e$, một sự mâu thuẫn, bởi $i + (t - j) < t$ nhưng t là giá trị dương bé nhất sao cho $a^{(t)} = e$. Do đó, mỗi thành phần của dãy $a^{(1)}, a^{(2)},..., a^{(t)}$ là riêng biệt, và $|<a>| = t$.

Hệ luận 33.18

Dãy $a^{(1)}, a^{(2)}$,...là tuần hoàn với chu kỳ t = ord(a); nghĩa là, $a^{(i)} = a^{(j)}$ nếu và chỉ nếu $i \equiv j \pmod t$.

Để nhất quán với hệ luận trên đây, ta định nghĩa $a^{(0)}$ dưới dạng e và $a^{(i)}$ dưới dạng $a^{(i \bmod t)}$ với tất cả các số nguyên i.

Hệ luận 33.19

Nếu $(S, \oplus)$ là một nhóm hữu hạn với đồng nhất thức e, thì với tất cả $a \in S$.

$a^{|S|} = e$.

Chứng minh Định lý Lagrange hàm ý rằng ord(a) | $|S|$, và do đó $|S|$ $\equiv$ 0 (mod t), ở đó t = ord(a).

Bài tập

33.3-1

Vẽ các bảng phép toán nhóm cho các nhóm $(\mathbf{Z}_4, +_4)$ và $(\mathbf{Z}^*_5, \cdot_5)$. Chứng tỏ các nhóm này là đẳng cấu bằng cách biểu lộ một tương ứng một-một α giữa các thành phần của chúng sao cho $a + b \equiv c$ (mod 4) nếu và chỉ nếu $\alpha(a) \cdot \alpha(b) \equiv \alpha(c)$ (mod 5).

33.3-2

Chứng minh Định lý 33.14.

33.3-3

Chứng tỏ nếu p là số nguyên tố và e là một số nguyên dương, thì

$\phi(p^e) = p^{e-1}(p - 1)$.

33.3-4

Chứng tỏ với bất kỳ $n > 1$ và với bất kỳ $a \in \mathbf{Z}_n^*$, hàm $f_a : \mathbf{Z}^*_n \to \mathbf{Z}^*_n$ được định nghĩa bởi $f_a(x) = ax \bmod n$ là một phép hoán vị của $\mathbf{Z}^*_n$.

33.3-5

Liệt kê tất cả các nhóm con của $\mathbf{Z}_9$ và của $\mathbf{Z}^*_{13}$.

33.4 Giải các phương trình tuyến tính modula

Giờ đây ta xét bài toán tìm các giải pháp cho phương trình

$$ax \equiv b \pmod{n}, \tag{33.22}$$

ở đó $n > 0$, một bài toán thực tiễn quan trọng. Ta mặc nhận có a, b, và n, và ta sẽ tìm các x, modulo n, thỏa phương trình (33.22). Có thể có zero, một , hoặc nhiều giải pháp như vậy.

Cho $\langle a \rangle$ thể hiện nhóm con của $\mathbf{Z}_n$ được phát sinh bởi a. Bởi $\langle a \rangle = \{a^{(x)} : x > 0\} = \{ax \bmod n : x > 0\}$, nên phương trình (33.22) có một giải pháp nếu và chỉ nếu $b \in \langle a \rangle$. Định lý Lagrange (Định lý 33.15) cho ta biết $|\langle a \rangle|$ phải là một ước số của n. Định lý dưới đây cho ta một mô tả đặc điểm chính xác của $\langle a \rangle$.

Định lý 33.20

Với bất kỳ số nguyên dương a và n, nếu $d = \gcd(a, n)$, thì

$$\langle a\rangle = \langle d\rangle = \{0, d, 2d, ..., ((n/d) - 1)d\}, \quad (33.23)$$

và như vậy

$$|\langle a\rangle| = n/d.$$

Chứng minh Ta bắt đầu bằng cách chứng tỏ $d \in \langle a\rangle$. Nhận thấy EXTENDED-EUCLID(a, n) tạo ra các số nguyên x' và y' sao cho $ax' + ny' = d$. Như vậy, $ax' \equiv d \pmod n$, sao cho $d \in (a)$.

Bởi $d \in \langle a\rangle$, dẫn đến mọi bội của d thuộc về $\langle a\rangle$, bởi vì một bội của một bội của a là một bội của a. Như vậy, $\langle a\rangle$ chứa mọi thành phần trong $\{0, d, 2d,...,((n/d) - 1)d\}$. Nghĩa là, $\langle d\rangle \subseteq \langle a\rangle$.

Giờ đây ta chứng tỏ $\langle a\rangle \subseteq \langle d\rangle$. Nếu $m \in \langle a\rangle$, thì $m = ax \bmod n$ với một số nguyên x, và do đó $m = ax + ny$ với một số nguyên y. Tuy nhiên, $d \mid a$ và $d \mid n$, và do đó $d \mid m$ theo phương trình (33.6). Như vậy, $m \in \langle d\rangle$.

Tổ hợp các kết quả này, ta có $\langle a\rangle = \langle d\rangle$. Để xem $|\langle a\rangle| = n/d$, nhận thấy có chính xác n/d bội của d giữa 0 và $n - 1$, kể cả các giá trị này.

Hệ luận 33.21

Phương trình $ax \equiv b \pmod n$ có thể giải được cho ẩn số x nếu và chỉ nếu $\gcd(a, n) \mid b$.

Hệ luận 33.22

Phương trình $ax \equiv b \pmod n$ có d các nghiệm riêng biệt modulo n, ở đó $d = \gcd(a, n)$, hoặc nó không có các nghiệm.

Chứng minh Nếu $ax \equiv b \pmod n$ có một nghiệm, thì $b \in \langle a\rangle$. Dãy $ai \bmod n$, với $i = 0, 1,...$ là tuần hoàn với chu kỳ $|\langle a\rangle| = n/d$, theo Hệ luận 33.18. Nếu $b \in \langle a\rangle$, thì b xuất hiện chính xác d lần trong dãy $ai \bmod n$, với $i = 0, 1,..., n - 1$, bởi khối các giá trị $\langle a\rangle$ có chiều dài-(n/d) được lặp lại chính xác d lần khi i gia tăng từ 0 đến $n - 1$. Các chỉ số x của các vị trí d này là các nghiệm của phương trình $ax \equiv b \pmod n$.

Định lý 33.23

Cho $d = \gcd(a, n)$, và giả sử rằng $d = ax' + ny'$ với một số nguyên x' và y' (ví dụ, như được tính toán bởi EXTENDED-EUCLID). Nếu $d \mid b$, thì phương trình $ax \equiv b \pmod n$ có giá trị x_0 làm một trong các nghiệm của nó, ở đó

$$x_0 = x'(b/d) \bmod n.$$

Chứng minh Bởi $ax' \equiv d \pmod n$, nên ta có

$$\begin{aligned} ax_0 &\equiv ax'(b/d) \pmod n \\ &\equiv d(b/d) \pmod n \\ &\equiv b \pmod n\,. \end{aligned}$$

và như vậy x_0 là một giải pháp cho $ax \equiv b \pmod n$.

Định lý 33.24

Giả sử rằng phương trình $ax \equiv b \pmod n$ giải được (nghĩa là, $d \mid b$, ở đó $d = \gcd(a, n)$) và x_0 là một nghiệm bất kỳ cho phương trình này. Như vậy, phương trình này có chính xác d nghiệm riêng biệt, modulo n, căn cứ vào $x_i = x_0 + i(n/d)$ với $i = 1, 2, ..., d-1$.

Chứng minh Bởi $n/d > 0$ và $0 \le i(n/d) < n$ với $i = 0, 1, ..., d-1$, tất cả các giá trị $x_0, x_1, ..., x_{d-1}$ là riêng biệt, modulo n. Theo tính chu kỳ của dãy $ai \bmod n$ (Hệ luận 33.18), nếu x_0 là một nghiệm của $ax \equiv b \pmod n$, thì mọi x_i là một nghiệm. Theo Hệ luận 33.22, có chính xác d nghiệm, sao cho $x_0, x_1, ..., x_{d-1}$ phải là tất cả của chúng.

Giờ đây ta đã phát triển toán học cần thiết để giải phương trình $ax \equiv b \pmod n$; thuật toán dưới đây in tất cả các nghiệm cho phương trình này. Các đầu vào a và b là các số nguyên tùy ý, và n là một số nguyên dương tùy ý.

```
MODULAR-LINEAR-EQUATION-SOLVER(a, b, n)
1  (d, x', y') ← EXTENDED-EUCLID (a, n)
2  if d | b
3      then x0 ← x'(b/d) mod n
4           for i ← 0 to d – 1
5               do print (x0 + i(n/d)) mod n
6      else print "không có các nghiệm"
```

Để lấy ví dụ về phép toán của thủ tục này, ta xét phương trình $14x \equiv 30 \pmod{100}$ (ở đây, $a = 14$, $b = 30$, và $n = 100$). Gọi EXTENDED-EUCLID trong dòng 1, ta được $(d, x, y) = (2, -7, 1)$. Bởi $2 \mid 30$, nên các dòng 3-5 được thi hành. Trong dòng 3, ta tính toán $x_0 = (-7)(15) \bmod 100 = 95$. Vòng lặp trên các dòng 4-5 sẽ in hai giải pháp: 95 và 45.

Thủ tục MODULAR-LINEAR-EQUATION-SOLVER làm việc như sau. Dòng 1 tính toán $d = \gcd(a, n)$ cũng như hai giá trị x' và y' sao cho $d = ax' + ny'$, chứng minh rằng x' là một nghiệm cho phương trình $ax' \equiv d \pmod n$. Nếu d không chia hết b, thì phương trình $ax \equiv b \pmod n$ không có nghiệm, theo Hệ luận 33.21. Dòng 2 kiểm tra xem $d \mid b$ hay không; nếu không, dòng 6 báo cáo không có các nghiệm. Bằng không,

dòng 3 tính toán một nghiệm x_0 cho phương trình (33.22), theo Định lý 33.23. Căn cứ vào một nghiệm, Định lý 33.24 phát biểu rằng $d - 1$ nghiệm khác có thể có được bằng cách cộng các bội của (n/d), modulo n. Vòng lặp **for** của các dòng 4-5 sẽ in ra tất cả d nghiệm, bắt đầu bằng x_0 và để (n/d) cách rời, modulo n.

Thời gian thực hiện của MODULAR-LINEAR-EQUATION-SOLVER là $O(\lg n + \gcd(a,n))$ phép toán số học, bởi EXTENDED-EUCLID chiếm $O(\lg n)$ phép toán số học, và mỗi lần lặp lại của vòng lặp **for** của các dòng 4-5 sẽ chiếm một số lượng bất biến các phép toán số học.

Các hệ luận dưới đây của Định lý 33.24 cho các điểm chuyên môn hóa đáng quan tâm.

Hệ luận 33.25

Với bất kỳ $n > 1$, nếu $\gcd(a, n) = 1$, thì phương trình $ax \equiv b \pmod{n}$ có một nghiệm duy nhất modulo n.

Nếu $b = 1$, một trường hợp chung đáng quan tâm, x mà ta đang tìm là một ***nghịch đảo nhân*** của a, modulo n.

Hệ luận 33.26

Với bất kỳ $n > 1$, nếu $\gcd(a, n) = 1$, thì phương trình

$$ax \equiv 1 \pmod{n} \qquad (33.24)$$

có một nghiệm duy nhất, modulo n. Bằng không, nó không có nghiệm.

Hệ luận 33.26 cho phép ta sử dụng hệ ký hiệu $(a^{-1} \bmod n)$ để tham chiếu nghịch đảo nhân của a, modulo n, khi a và n là nguyên tố cùng nhau. Nếu $\gcd(a, n) = 1$, thì một nghiệm cho phương trình $ax \equiv 1 \pmod{n}$ là số nguyên x được trả về bởi EXTENDED-EUCLID, vì phương trình

$$\gcd(a, n) = 1 = ax + ny$$

hàm ý $ax \equiv 1 \pmod{n}$. Như vậy, $(a^{-1} \bmod n)$ có thể được tính toán một cách hiệu quả bằng EXTENDED-EUCLID.

Bài tập

33.4-1

Tìm tất cả các nghiệm cho phương trình $35x \equiv 10 \pmod{50}$.

33.4-2

Chứng minh phương trình $ax \equiv ay \pmod{n}$ hàm ý $x \equiv y \pmod{n}$ mỗi khi $\gcd(a, n) = 1$. Chứng tỏ điều kiện $\gcd(a, n) = 1$ là cần thiết bằng cách cung cấp một ví dụ ngược với $\gcd(a, n) > 1$.

33.4-3

Xét thay đổi dưới đây cho dòng 3 của MODULAR-LINEAR-EQUATION-SOLVER:

3 **then** $x_0 \leftarrow x'(b/d) \bmod (n/d)$

Điều này có làm việc không? Giải thích tại sao có hoặc tại sao không.

33.4-4 *

Cho $f(x) \equiv f_0 + f_1 x + ... + f_t x^t \pmod p$ là một đa thức có bậc t, với các hệ số f_i được rút từ $\mathbf{Z}_p$, ở đó p là số nguyên tố. Ta nói rằng một $a \in \mathbf{Z}_p$ và một ***zero*** của f nếu $f(a) \equiv 0 \pmod p$. Chứng minh rằng nếu a là một zero của f, thì $f(x) \equiv (x - a)g(x) \pmod p$ với một đa thức $g(x)$ có bậc $t - 1$. Bằng phương pháp quy nạp trên t, chứng minh rằng một đa thức $f(x)$ có bậc t có thể có tối đa t zero riêng biệt modulo một số nguyên tố p.

33.5 Định lý phần dư Trung Quốc

Vào khoảng năm 100 A.D., nhà toán học Trung quốc Sun-Tsu đã giải bài toán tìm các số nguyên x để lại các số dư 2, 3, và 2 khi được chia cho 3, 5, và 7 theo thứ tự nêu trên. Một giải pháp như vậy đó là $x = 23$; tất cả các nghiệm đều có dạng $23 + 105k$ với các số nguyên k tùy ý. "Định lý phần dư Trung Quốc" cung cấp một dạng tương ứng giữa một hệ thống các phương trình modulo một tập hợp các moduli nguyên tố cùng nhau theo từng cặp (ví dụ, 3, 5, và 7) và một phương trình modulo tích của chúng (ví dụ, 105).

Định lý phần dư Trung Quốc có hai công dụng chính. Cho số nguyên $n = n_1 n_2 ... n_k$, ở đó các thừa số n_i là nguyên tố cùng nhau theo từng cặp. Trước tiên, định lý phần dư Trung Quốc là một "định lý cấu trúc" mô tả mô tả cấu trúc của $\mathbf{Z}_n$ là đồng nhất với của tích Đề các $\mathbf{Z}_{n_1} \times \mathbf{Z}_{n_2} \times ... \times \mathbf{Z}_{n_k}$ với phép cộng và phép nhân theo từng thành phần modulo n_i trong thành phần thứ i. Thứ hai, mô tả này thường được dùng để cho ra các thuật toán hiệu quả, bởi làm việc trong mỗi trong số các hệ thống $\mathbf{Z}_{n_i}$ có thể hiệu quả hơn (theo dạng các phép toán bit) so với làm việc modulo n.

Định lý 33.27 (Định lý phần dư Trung Quốc)

Cho $n = n_1 n_2 ... n_k$, ở đó n_i là nguyên tố cùng nhau theo từng cặp. Xét dạng tương ứng

$$a \leftrightarrow (a_1, a_2, ..., a_k), \tag{33.25}$$

ở đó $a \in \mathbf{Z}_n$, $a_i \in \mathbf{Z}_{n_i}$, và

$a_i = a \bmod n_i$

với $i = 1, 2, ..., k$. Như vậy, phép ánh xạ (33.25) là một tương ứng một-một (phép song ánh) giữa $\mathbf{Z}_n$ và tích Đề các $\mathbf{Z}_{n_1} \times \mathbf{Z}_{n_2} \times \ldots \times \mathbf{Z}_{n_k}$. Các phép toán được thực hiện trên các thành phần của $\mathbf{Z}_n$ có thể được thực hiện tương đương trên bộ k tương ứng bằng cách thực hiện các phép toán độc lập trong mỗi vị trí tọa bậc trong hệ thống thích hợp. Nghĩa là, nếu

$a \leftrightarrow (a_1, a_2, ..., a_k)$,

$b \leftrightarrow (b_1, b_2, ..., b_k)$,

thì

$$(a + b) \bmod n \leftrightarrow ((a_1 + b_1) \bmod n_1, ..., (a_k + b_k) \bmod n_k) , \quad (33.26)$$

$$(a - b) \bmod n \leftrightarrow ((a_1 - b_1) \bmod n_1, ..., (a_k - b_k) \bmod n_k) , \quad (33.27)$$

$$(ab) \bmod n \leftrightarrow (a_1 b_1 \bmod n_1, ... , a_k b_k \bmod n_k) . \quad (33.28)$$

Chứng minh Việc biến đổi giữa hai phép biểu diễn là khá đơn giản. Đi từ a đến $(a_1 , a_2, ..., a_k)$ chỉ yêu cầu k phép chia. Việc tính toán a từ các đầu vào $(a_1, a_2, ..., a_k)$ cũng dễ không kém, dùng công thức dưới đây. Cho $m_i = n/n_i$ với $i = 1, 2, ..., k$. Lưu ý, $m_i = n_1 n_2 ... n_{i-1} n_{i+1} ... n_k$, sao cho $m_i \equiv 0 \pmod{n_j}$ với tất cả $j \neq i$. Như vậy, cho

$$c_i = m_i(m^{-1}_i \bmod n_i) \quad (33.29)$$

với $i = 1, 2, ..., k$, ta có

$$a \equiv (a_1 c_1 + a_2 c_2 + ... + a_k c_k) \pmod n . \quad (33.30)$$

Phương trình 33.29 được định nghĩa kỹ, bởi m_i và n_i là nguyên tố cùng nhau (theo Định lý 33.6), và do đó Hệ luận 33.26 hàm ý rằng $(m^{-1}_i \bmod n_i)$ được định nghĩa. Để xác minh phương trình (33.30), lưu ý rằng $c_j \equiv m_j \equiv 0 \pmod{n_i}$ nếu $j \neq i$, và rằng $c_i \equiv 1 \pmod{n_i}$. Như vậy, ta có phép tương ứng

$c_i \leftrightarrow (0, 0, ..., 0, 1, 0, ..., 0)$,

một vectơ có các 0 ở mọi nơi ngoại trừ trong tọa bậc thứ i, ở đó nó có một 1. Như vậy, c_i hình thành một "cơ sở" cho phần biểu diễn, theo một nghĩa nào đó.

Do đó, với mỗi i, ta có

$$\begin{aligned} a &\equiv a_i c_i && \pmod{n_i} \\ &\equiv a_i m_i(m^{-1}_i \bmod n_i) && \pmod{n_i} \end{aligned}$$

$\equiv \quad a_i \qquad\qquad (\text{mod } n_i)$.

Bởi ta có thể biến đổi theo cả hai hướng, nên phép tương ứng là một-một. Các phương trình (33.26)-(33.28) trực tiếp đến từ Bài tập 33.1-6, bởi x mod n_i = (x mod n) mod n_i với bất kỳ x và i = 1, 2, ..., k.

Các hệ luận dưới đây sẽ được dùng về sau trong chương này.

Hệ luận 33.28

Nếu $n_1, n_2, ..., n_k$ là nguyên tố cùng nhau theo từng cặp và $n = n_1 n_2 ... n_k$, thì với bất kỳ số nguyên $a_1, a_2, ..., a_k$, tập hợp các phương trình đồng thời

$x \equiv a_i \ (\text{mod } n_i)$,

với i = 1, 2,..., k, có một nghiệm duy nhất modulo n với ẩn số x.

Hệ luận 33.29

Nếu $n_1, n_2,..., n_k$ là nguyên tố cùng nhau theo từng cặp và $n = n_1 n_2 ... n_k$, thì với tất cả các số nguyên x và a,

$x \equiv a \ (\text{mod } n_i)$

với i = 1, 2,..., k nếu và chỉ nếu

$x \equiv a \ (\text{mod } n)$.

Để lấy ví dụ về định lý phần dư Trung Quốc, giả sử ta có hai phương trình

$a \equiv 2 \quad (\text{mod } 5)$,

$a \equiv 3 \quad (\text{mod } 13)$,

sao cho $a_1 = 2$, $n_1 = m_2 = 5$, $a_2 = 3$, và $n_2 = m_1 = 13$, và ta muốn tính toán a mod 65, bởi $n = 65$. Bởi vì $13^{-1} \equiv 2 \ (\text{mod } 5)$ và $5^{-1} \equiv 8 \ (\text{mod } 13)$, ta có

$c_1 = 13(2 \text{ mod } 5) = 26$,

$c_2 = 5(8 \text{ mod } 13) = 40$,

và

$$
\begin{aligned}
a &\equiv 2 \cdot 26 + 3 \cdot 40 && (\text{mod } 65) \\
&\equiv 52 + 120 && (\text{mod } 65) \\
&\equiv 42 && (\text{mod } 65) .
\end{aligned}
$$

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	40	15	55	30	5	45	20	60	35	10	50	25
1	26	1	41	16	56	31	6	46	21	61	36	11	51
2	52	27	2	42	17	57	32	7	47	22	62	37	12
3	13	53	28	3	43	18	58	33	8	48	23	63	38
4	39	14	54	29	4	44	19	59	34	9	49	24	64

Hình 33.3 Một minh họa của định lý phần dư Trung Quốc với $n_1 = 5$ và $n_2 = 13$. Với ví dụ này, $c_1 = 26$ và $c_2 = 40$. Trong hàng i, cột j được nêu giá trị của a, modulo 65, sao cho (a mod 5) = i và (a mod 13) = j. Lưu ý, hàng 0, cột 0 chứa một 0. Cũng vậy, hàng 4, cột 12 chứa một 64 (tương đương với -1). Bởi $c_1 = 26$, việc dời xuống một hàng sẽ làm tăng a lên 26. Cũng vậy, $c_2 = 40$ có nghĩa là việc dời phải một cột sẽ làm tăng a lên 40. Tăng a lên 1 tương ứng với việc dời chéo đổ xuống và sang phải, đóng khung vòng quanh từ dưới lên và từ phải qua trái.

Xem Hình 33.3 để có minh họa của định lý phần dư Trung Quốc, modulo 65.

Như vậy, ta có thể làm việc modulo n bằng cách làm việc modulo n trực tiếp hoặc làm việc trong phần biểu diễn được biến đổi dùng các phép tính modulo n_i riêng biệt, nếu thấy tiện. Các phép tính hoàn toàn tương đương.

Bài tập

33.5-1

Tìm tất cả đến nghiệm cho các phương trình $x \equiv 4 \pmod 5$ và $x \equiv 5 \pmod{11}$.

33.5-2

Tìm tất cả các số nguyên x để lại các số dư 1, 2, 3, 4, 5 khi được chia cho 2, 3, 4, 5, 6, theo thứ tự nêu trên.

33.5-3

Chứng tỏ, dưới các phần định nghĩa của Định lý 33.27, nếu $\gcd(a, n) = 1$, thì $(a^{-1} \bmod n) \leftrightarrow ((a_1^{-1} \bmod n_1), (a_2^{-1} \bmod n_2), \ldots, (a_k^{-1} \bmod n_k))$.

33.5-4

Dưới các phần định nghĩa của Định lý 33.27, chứng minh số lượng các căn của phương trình $f(x) \equiv 0 \pmod n$ bằng với tích của số lượng các căn một mỗi phương trình $f(x) \equiv 0 \pmod{n_1}$, $f(x) \equiv 0 \pmod{n_2}$, ..., $f(x) \equiv 0 \pmod{n_k}$.

33.6 Các lũy thừa của một thành phần

Tự nhiên giống như khi xét các bội của một thành phần đã cho a, modulo n, ta cảm thấy tự nhiên khi xét dãy các lũy thừa của a, modulo n, ở đó $a \in \mathbf{Z}^*_n$:

$$a^0, a^1, a^2, a^3, \ldots, \qquad (33.31)$$

modulo n. Lập chỉ số từ 0, giá trị thứ 0 trong dãy này là $a^0 \bmod n = 1$, và giá trị thứ i là $a^i \bmod n$. Ví dụ, các lũy thừa của 3 modulo 7 là

i	0	1	2	3	4	5	6	7	8	9	10	11	...
$3^i \bmod 7$	1	3	2	6	4	5	1	3	2	6	4	5	...

trong khi đó các lũy thừa của 2 modulo 7 là

i	0	1	2	3	4	5	6	7	8	9	10	11	...
$2^i \bmod 7$	1	2	4	1	2	4	1	2	4	1	2	4	...

Trong đoạn này, cho $\langle a \rangle$ thể hiện nhóm con của $\mathbf{Z}^*_n$ do a phát sinh, và cho $\text{ord}_n\langle a \rangle$ ("thứ tự của a, modulo n") thể hiện thứ tự của a trong $\mathbf{Z}^*_n$. Ví dụ, $\langle 2 \rangle = \{1, 2, 4\}$ trong $\mathbf{Z}^*_7$, và $\text{ord}_7 (2) = 3$. Dùng phần định nghĩa của hàm phi của Euler $\phi(n)$ làm kích cỡ của $\mathbf{Z}^*_n$ (xem Đoạn 33.3), giờ đây ta phiên dịch Hệ luận 33.19 thành hệ ký hiệu của $\mathbf{Z}^*_n$ để được định lý Euler và chuyên hóa nó theo $\mathbf{Z}^*_p$, ở đó p là số nguyên tố, để có được định lý Fermat.

Định lý 33.30 (định lý Euler)

Với bất kỳ số nguyên $n > 1$,

$$a^{\phi(n)} \equiv 1 \pmod{n} \text{ với tất cả } a \in \mathbf{Z}^*_n. \qquad (33.32)$$

Định lý 33.31 (định lý Fermat)

Nếu p là số nguyên tố, thì

$$a^{p-1} \equiv 1 \pmod{p} \text{ với tất cả } a \in \mathbf{Z}^*_p. \qquad (33.33)$$

Chứng minh Theo phương trình (33.21), $\phi(p) = p - 1$ nếu p là số nguyên tố.

Hệ luận này áp dụng cho mọi thành phần trong $\mathbf{Z}_p$ ngoại trừ 0, bởi $0 \notin \mathbf{Z}^*_p$. Tuy nhiên, với tất cả $a \in \mathbf{Z}_p$, ta có $a^p \equiv a \pmod{p}$ nếu p là số nguyên tố.

Nếu $\text{ord}_n (g) = |\mathbf{Z}^*_n|$, thì mọi thành phần trong $\mathbf{Z}^*_n$ là một lũy thừa của g, modulo n, và ta nói rằng g là một ***căn nguyên thủy*** hoặc một ***bộ sinh*** của $\mathbf{Z}^*_n$. Ví dụ, 3 là một căn nguyên thủy, modulo 7. Nếu $\mathbf{Z}^*_n$, có một căn nguyên thủy, ta nói rằng nhóm $\mathbf{Z}^*_n$ là chu trình. Ta bỏ qua chứng

minh của định lý dưới đây, đã được Niven và Zuckerman [151] chứng minh.

Định lý 33.32

Các giá trị của $n > 1$ mà $\mathbf{Z}^*_n$ là chu trình là 2, 4, p^e, và $2p^e$, với tất cả các số nguyên tố lẻ p và tất cả các số nguyên dương e.

Nếu g là một căn nguyên thủy của $\mathbf{Z}^*_n$, và a là bất kỳ thành phần nào của $\mathbf{Z}^*_n$, thì ở đó tồn tại một z sao cho $g^z \equiv a \pmod n$. z này được gọi là *lôga rời rạc* hoặc *chỉ số* của a, modulo n, đến cơ số g; ta thể hiện giá trị này dưới dạng $\mathrm{ind}_{n,g}(a)$.

Định lý 33.33 (Định lý lôga rời rạc)

Nếu g là một căn nguyên thủy của $\mathbf{Z}^*_n$, thì phương trình $g^x \equiv g^y \pmod n$ tồn tại nếu và chỉ nếu phương trình $x \equiv y \pmod{\phi(n)}$ tồn tại.

Chứng minh Trước tiên giả sử rằng $x \equiv y \pmod{\phi(n)}$. Như vậy, $x = y + k\phi(n)$ với một số nguyên k. Do đó,

$$
\begin{aligned}
g^x &\equiv g^{y+k\phi(n)} && \pmod n \\
&\equiv g^y \bullet (g^{\phi(n)})^k && \pmod n \\
&\equiv g^y \bullet 1^k && \pmod n \\
&\equiv g^y && \pmod n
\end{aligned}
$$

Ngược lại, giả sử $g^x \equiv g^y \pmod n$. Bởi dãy các lũy thừa của g phát sinh mọi thành phần của $\langle g\rangle$ và $|\langle g\rangle| = \phi(n)$, Hệ luận 33.18 hàm ý rằng dãy các lũy thừa của g là tuần hoàn với chu kỳ $\phi(n)$. Do đó, nếu $g^x \equiv g^y \pmod n$, thì ta phải có $x \equiv y \pmod{\phi(n)}$.

Việc lấy lôga rời rạc đôi lúc có thể rút gọn việc biện luận về một phương trình, như minh họa trong phần chứng minh của định lý dưới đây.

Định lý 33.34

Nếu p là một số nguyên tố lẻ và $e \geq 1$, thì phương trình

$$x^2 \equiv 1 \pmod{p^e} \tag{33.34}$$

chỉ có hai nghiệm, tức là $x = 1$ và $x = -1$.

Chứng minh Cho $n = p^e$. Định lý 33.32 hàm ý rằng $\mathbf{Z}^*_n$ có một căn nguyên thủy g. Phương trình (33.34) có thể được viết

$$(g^{\mathrm{ind}_{n,g}(x)})^2 \equiv g^{\mathrm{ind}_{n,g}(1)} \pmod n\,. \tag{33.35}$$

Sau khi lưu ý rằng $\mathrm{ind}_{n,g}(1) = 0$, ta nhận thấy Định lý 33.33 hàm ý phương trình (33.35) tương đương với

$$2 \, . \, \mathrm{ind}_{n,g}(x) \equiv 0 \pmod{\phi(n)} \, . \qquad (33.36)$$

Để giải phương trình này với ẩn số $\mathrm{ind}_{n,g}(x)$, ta áp dụng các phương pháp của Đoạn 33.4. Cho $d = \gcd(2, \phi(n)) = \gcd(2, (p-1)p^{e-1}) = 2$, và lưu ý $d \mid 0$, từ Định lý 33.24 ta thấy rằng phương trình (33.36) có chính xác $d = 2$ nghiệm. Do đó, phương trình (33.34) có chính xác 2 nghiệm, là $x = 1$ và $x = -1$ theo thẩm tra.

Một số x là một ***căn bậc hai không hiển nhiên của 1, modulo n,*** nếu nó thỏa phương trình $x^2 \equiv 1 \pmod n$ nhưng x không tương đương với một trong hai căn bậc hai "hiển nhiên": 1 hoặc -1, modulo n. Ví dụ, 6 là một căn bậc hai không hiển nhiên của 1, modulo 35. Hệ luận dưới đây đối với Định lý 33.34 sẽ được dùng để chứng minh tính đúng đắn cho thủ tục thử tính nguyên Miller-Rabin trong Đoạn 33.8.

Hệ luận 33.35

Nếu ở đó tồn tại một căn bậc hai không hiển nhiên của 1, modulo n, thì n là hợp số.

Chứng minh Hệ luận này chẳng qua tương phản với Định lý 33.34. Nếu ở đó tồn tại một căn bậc hai không hiển nhiên của 1, modulo n, thì n không thể là một số nguyên tố hoặc một lũy thừa của một số nguyên tố.

Nâng lên các lũy thừa với phép bình phương lặp lại

Một phép toán thường xảy ra trong các phép tính lý thuyết số đó là nâng một số thành một lũy thừa modulo một số khác, còn gọi là ***mũ hóa modula***. Một cách chính xác hơn, ta muốn một cách hiệu quả để tính toán $a^b \bmod n$, ở đó a và b là các số nguyên không âm và n là một số nguyên dương. Mũ hóa modula cũng là một phép toán chủ yếu trong nhiều thường trình thử tính nguyên [primality-testing routines] và trong hệ mật mã khóa công RSA. Phương pháp của ***phép bình phương lặp lại*** giải bài toán này một cách hiệu quả nhờ dùng phần biểu diễn nhị phân của b.

Cho $\langle b_k, b_{k-1}, \ldots, b_1, b_0 \rangle$ là phần biểu diễn nhị phân của b. (Nghĩa là, phần biểu diễn nhị phân dài $k + 1$ bit, b_k là bit quan trọng nhất, và b_0 là bit ít quan trọng nhất.) Thủ tục dưới đây tính toán $a^c \bmod n$ khi c được tăng bằng các lần nhân đôi và các lần gia số từ 0 đến b.

MODULAR-EXPONENTIATION(a, b, n)

1 $c \leftarrow 0$

2 $d \leftarrow 1$

3 cho $\langle b_k, b_{k-1}, \ldots, b_0 \rangle$ be the phần biểu diễn nhị phân of b

```
for i ← k downto 0
    do c ← 2c
       d ← (d • d) mod n
       if b_i = 1
          then c ← c + 1
               d ← (d • a) mod n
return d
```

i	9	8	7	6	5	4	3	2	1	0
b_i	1	0	0	0	1	1	0	0	0	0
c	1	2	4	8	17	35	70	140	280	560
d	7	49	157	526	160	241	298	166	67	1

Hình 33.4 Các kết quả của MODULAR-EXPONENTIATION khi tính toán a^b (mod n), ở đó $a = 7$, $b = 560 = \langle 1000110000 \rangle$, và $n = 561$. Các giá trị được nêu sau mỗi lần thi hành của vòng lặp **for**. Kết quả cuối cùng là 1.

Mỗi số mũ được tính toán trong một dãy là gấp hai lần so với số mũ trước đó hoặc hơn một lần so với số mũ trước đó; phần biểu diễn nhị phân của b của đọc từ phải qua trái để điều khiển các phép toán sẽ được thực hiện. Mỗi lần lặp lại của vòng lặp sử dụng một trong các đồng nhất thức

$$a^{2c} \bmod n = (a^c)^2 \bmod n ,$$

$$a^{2c+1} \bmod n = a \bullet (a^c)^2 \bmod n ,$$

tùy thuộc vào việc $b_i = 0$ hay 1, theo thứ tự nêu trên. Công dụng chủ yếu của phép bình phương trong mỗi lần lặp lại đã giải thích tên gọi "phép bình phương lặp lại." Ngay sau khi bit b_i được đọc và xử lý, giá trị của c giống như tiền tố $\langle b_k, b_{k-1}, \ldots, b_i \rangle$ của phần biểu diễn nhị phân của b. Để lấy ví dụ, với $a = 7$, $b = 560$, và $n = 561$, thuật toán tính toán dãy các giá trị modulo 561 nêu trong Hình 33.4; dãy các số mũ đã dùng được nêu trong hàng c của bảng.

Thuật toán không thực sự cần biến c song nó được gộp để dễ bề giải thích: thuật toán bảo toàn sự bất biến rằng $d = a^c \bmod n$ khi nó làm tăng c theo các lần nhân đôi và gia số cho đến khi $c = b$. Nếu các đầu vào a, b, và n là các số β-bit, thì tổng số các phép toán số học cần có là $O(\beta)$ và tổng các phép toán bit cần có là $O(\beta^3)$.

Bài tập

33.6-1

Vẽ một bảng nêu thứ tự mọi thành phần trong $\mathbf{Z}^*_{11}$. Lấy căn nguyên thủy nhỏ nhất g và tính toán một bảng cho ra $\text{ind}_{11,g}(x)$ với tất cả $x \in \mathbf{Z}^*_{11}$.

33.6-2

Nêu một thuật toán mũ hóa modula xét các bit của b từ phải qua trái thay vì trái qua phải.

33.6-3

Giải thích cách tính toán $a^{-1} \bmod n$ với bất kỳ $a \in \mathbf{Z}^*_n$ dùng thủ tục MODULAR-EXPONENTIATION, giả định bạn biết $\phi(n)$.

33.7 Hệ mật mã khóa công RSA

Một hệ mật mã khóa công có thể được dùng để mã hóa các thông điệp gửi đi giữa hai bên truyền thông sao cho bọn nghe trộm tình cờ nghe được các thông điệp đã mã hóa sẽ không thể giải mã chúng. Một hệ mật mã khóa công cũng cho phép một bên chấp một "chữ ký điện tử" (hoặc ký danh điện tử_ND) không thể giả mạo vào cuối của một thông điệp điện tử. Một chữ ký như vậy là phiên bản điện tử của một chữ ký viết tay trên một tư liệu giấy. Mọi người dễ dàng kiểm tra nó, không ai có thể giả mạo, song lại mất đi tính hợp lệ của nó nếu có một bit bất kỳ của thông điệp bị thay đổi. Do đó nó cung cấp khả năng thẩm định quyền về danh xưng của người ký lẫn nội dung của thông điệp đã ký. Nó là công cụ hoàn hảo cho các hợp đồng kinh doanh được ký bằng điện tử, các ngân phiếu điện tử, các đơn mua hàng điện tử, và các cuộc liên lạc điện tử khác phải được thẩm định quyền.

Hệ mật mã khóa công RSA dựa trên sự khác biệt quan trọng giữa việc dễ tìm các số nguyên tố lớn và việc khó lấy thừa số tích của hai số nguyên tố lớn. Đoạn 33.8 mô tả một thủ tục hiệu quả để tìm các số nguyên tố lớn, và Đoạn 33.9 mô tả bài toán lấy thừa số các số nguyên lớn.

Các hệ mật mã hóa khóa công

Trong một hệ mật mã hóa khóa công, mỗi bên tham gia đều có một ***khóa công*** [public key] và một ***khóa bí mật*** [secret key]. Mỗi khóa là một mẩu thông tin. Ví dụ, trong hệ mật mã RSA, mỗi khóa bao gồm một cặp số nguyên. Theo truyền thống, các bên tham gia "Alice" và "Bob"

được dùng trong các ví dụ về mật mã; ta thể hiện các khóa công và tư của họ dưới dạng P_A, S_A cho Alice và P_B, S_B cho Bob.

Mỗi bên tham gia tạo các khóa công và bí mật riêng. Mỗi bên giữ khóa bí mật của mình thật bí mật, nhưng họ có thể biểu lộ công khóa của mình cho bất kỳ ai hoặc thậm chí công khai nó. Thực vậy, để tiện dụng, ta mặc nhận rằng khóa công của mọi người đều sẵn có trong một thư mục công, sao cho bất kỳ bên tham gia nào cũng có thể dễ dàng có được khóa công của bất kỳ bên tham gia khác.

Các khóa công và bí mật chỉ định các hàm có thể áp dụng cho bất kỳ thông điệp nào. Cho $\mathcal{D}$ thể hiện tập hợp các thông điệp chấp nhận được. Ví dụ, $\mathcal{D}$ có thể là tập hợp tất cả các dãy bit có chiều dài hữu hạn. Ta yêu cầu các khóa công và bí mật chỉ định các hàm từ $\mathcal{D}$ đến chính nó. Hàm tương ứng với khóa công P_A của Alice được ký hiệu là $P_A()$, và hàm tương ứng với khóa bí mật S_A của cô được ký hiệu là $S_A()$. Như vậy, các hàm $P_A()$ và $S_A()$ là các phép hoán vị của $\mathcal{D}$. Ta mặc nhận rằng các hàm $P_A()$ và $S_A()$ có thể tính toán hiệu quả căn cứ vào khóa tương ứng P_A hoặc S_A.

Các khóa công và bí mật của một bên tham gia bất kỳ là một "cặp ăn khớp" ở chỗ chúng chỉ định các hàm là các nghịch đảo với nhau. Nghĩa là,

$$M = S_A(P_A(M)), \tag{33.37}$$

$$M = P_A(S_A(M)) \tag{33.38}$$

với một thông điệp $M \in \mathcal{D}$. Việc biến đổi thành công M bằng hai khóa P_A và S_A, theo một trong hai thứ tự, sẽ cho ra thông điệp M trở lại.

Trong một hệ mật mã khóa công, nhất thiết không ai ngoài Alice có thể tính toán hàm $S_A()$ trong mọi thời lượng thực tiễn. Tính riêng tư của thư được mã hóa và gửi đến Alice và tính xác thực của các ký danh số hóa của Alice đều dựa vào giả thiết cho rằng chỉ mình Alice mới có thể tính toán $S_A()$. Yêu cầu này cho biết tại sao Alice phải giữ S_A bí mật; nếu không giữ, cô sẽ mất tính duy nhất của mình và hệ mật mã không thể cung cấp cho cô các khả năng duy nhất. Giả thiết cho rằng chỉ mình Alice mới có thể tính toán $S_A()$ phải đứng vững cho dù mọi người biết P_A và có thể tính toán $P_A()$, hàm nghịch đảo với $S_A()$, một cách hiệu quả. Khó khăn chính trong việc thiết kế một hệ mật mã khóa công hiệu quả đó là phát hiện cách tạo một hệ thống ở đó ta có thể biểu lộ một phép biến đổi $P_A()$ mà không biểu lộ cách tính toán phép biến đổi nghịch đảo tương ứng $S_A()$.

Trong một hệ mật mã khóa công, tiến trình mật mã hóa làm việc như sau. Giả sử

• Bob muốn gửi cho Alice một thông điệp M đã mã hóa sao cho nó trông giống như một dạng câu văn sai ngữ pháp không thể hiểu được đối với bọn nghe trộm. Bối cảnh để gửi thông điệp sẽ như sau.

• Bob có được khóa công P_A của Alice (từ một thư mục công hoặc trực tiếp từ Alice).

• Bob tính toán ***văn bản mật mã*** $C = P_A(M)$ tương ứng với thông điệp M và gửi C cho Alice.

• Khi Alice nhận văn bản mật mã C, mà cô áp dụng khóa bí mật SA của mình để truy lục thông điệp ban đầu: $M = S_A(C)$.

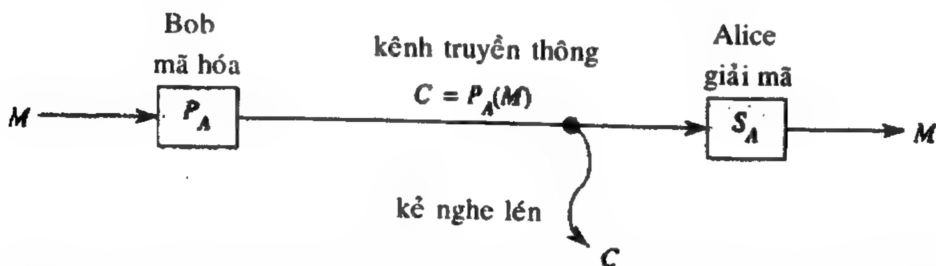
Hình 33.5 minh họa tiến trình này. Bởi $S_A()$ và $P_A()$ là các hàm nghịch đảo, nên Alice có thể tính toán M từ C. Bởi chỉ mình Alice có thể tính toán $S_A()$, nên chỉ Alice mới có thể tính toán M từ C. Việc mã hóa M dùng $P_A()$ đã bảo vệ M khỏi bị để lộ trước mọi người ngoại trừ Alice.

Các ký danh số hóa (hoặc chữ ký số hóa_ND) cũng dễ thực thi như trong một hệ mật mã hóa khóa công. Giờ đây, giả sử Alice muốn gửi cho Bob một hồi âm M' có ký nhận theo kỹ thuật số. Bối cảnh ký danh số hóa diễn tiến như sau.

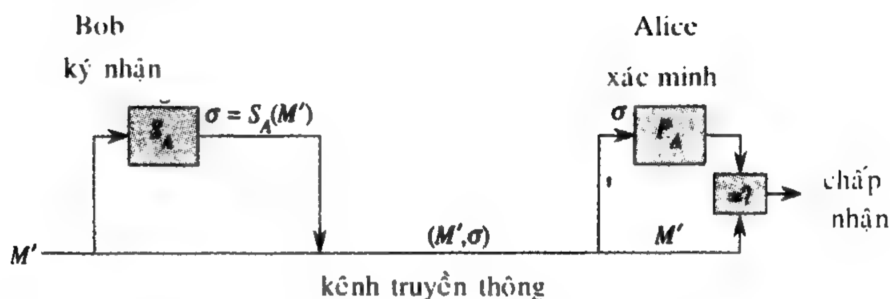
• Alice tính toán ***chữ ký số hóa*** σ của cô cho thông điệp M' dùng khóa bí mật S_A của cô và phương trình $\sigma = S_A(M')$.

• Alice gửi cặp thông điệp/chữ ký (M', σ) cho Bob.

• Khi Bob nhận được (M', σ), anh có thể xác minh nó đã xuất phát từ Alice dùng khóa công của Alice bằng cách xác minh phương trình $M' = P_A(\sigma)$. (Có lẽ, M' chứa tên của Alice, do đó Bob biết phải sử dụng khóa công của người nào.) Nếu phương trình đứng vững, Bob kết luận rằng thông điệp M' thực tế đã được Alice ký nhận. Nếu phương trình không đứng vững, Bob kết luận hoặc thông điệp M' hoặc chữ ký số hóa σ đã bị sai lạc bởi các lỗi truyền hoặc cặp (M', σ) là một dạng giả mạo để thử.



Hình 33.5 Tiến trình mật mã hóa trong một hệ thống khóa công. Bob mã hóa thông điệp M dùng khóa công P_A của Alice và truyền văn bản mật mã kết quả $C = P_A(M)$ cho Alice. Một kẻ nghe lén chốt giữ văn bản mật mã đã truyền không nắm được thông tin về M. Alice nhận C và giải mã nó dùng khóa bí mật của cô để có được thông điệp ban đầu $M = S_A(C)$.



Hình 33.6 Các ký danh số hóa trong một hệ thống khóa công. Alice ký nhận thông điệp M' bằng cách chấp ký danh số hóa $\sigma = S_A(M')$ vào nó. Cô truyền cặp thông điệp/chữ ký (M', σ) cho Bob, Bob xác minh nó bằng cách kiểm tra phương trình $M' = P_A(\sigma)$. Nếu phương trình đứng vững, anh chấp nhận (M', σ) là một thông điệp đã được Alice ký nhận.

Hình 33.6 minh họa tiến trình này. Bởi một chữ ký số hóa cung cấp cả sự thẩm định quyền danh xưng của người ký lẫn sự thẩm định quyền về nội dung của thông điệp đã ký, nên nó tương tự như một chữ ký viết tay ở cuối một tư liệu thành văn.

Một tính chất quan trọng của một chữ ký số hóa đó là nó có thể xác minh bởi bất kỳ ai có quyền truy cập khóa công của người ký. Một thông điệp đã ký có thể được xác minh bởi một bên rồi tiếp tục chuyền đi cho các bên khác cũng có thể xác minh chữ ký. Ví dụ, thông điệp có thể là một ngân phiếu điện tử từ Alice đến Bob. Sau khi Bob xác minh chữ ký của Alice trên ngân phiếu, anh có thể trao ngân phiếu cho ngân hàng của mình, sau đó ngân hàng cũng có thể xác minh chữ ký và thực hiện việc chuyển giao ngân quỹ thích hợp.

Ta lưu ý một thông điệp đã ký không được mã hóa; thông điệp ở dạng "trong sáng" (văn bản thường_ND) và không được bảo vệ khỏi bị để lộ. Nhờ soạn thảo các giao thức trên đây để mật mã hóa và ký nhận, ta có thể tạo ra các thông điệp vừa được ký nhận và được mã hóa. Trước tiên người ký chấp chữ ký số hóa của mình vào thông điệp rồi mã hóa cặp thông điệp/chữ ký kết quả bằng khóa công của người sẽ nhận. Người nhận giải mã thông điệp đã nhận bằng khóa bí mật của họ để có được cả thông điệp ban đầu lẫn chữ ký số hóa củanó. Sau đó, anh ta có thể xác minh chữ ký nhờ dùng khóa công của người ký. Theo các hệ thống gốc giấy tờ, tiến trình phối hợp tương ứng đó là ký nhận tư liệu giấy rồi niêm phong tư liệu bên trong một phong bì giấy mà chỉ người được đề nhận mới có quyền mở.

Hệ mật mã RSA

Trong ***hệ mật mã khóa công RSA***, một bên tham gia tạo các khóa

công và bí mật của mình theo thủ tục dưới đây.

1. Lựa chọn ngẫu nhiên hai số nguyên tố lớn p và q. Các số nguyên tố p và q có thể là, giả sử, 100 chữ số thập phân cho mỗi số.

2. Tính toán n theo phương trình $n = pq$.

3. Lựa chọn một số nguyên lẻ nhỏ e là nguyên tố cùng nhau với $\phi(n)$, mà , theo phương trình (33.20), sẽ bằng $(p-1)(q-1)$.

4. Tính toán d dưới dạng nghịch đảo nhân của e, modulo $\phi(n)$. (Hệ luận 33.26 bảo đảm d tồn tại và được định nghĩa duy nhất.)

5. Công bố cặp $P = (e, n)$ dưới dạng *khóa công RSA* của mình.

6. Giữ bí mật cặp $S = (d, n)$ làm *khóa bí mật RSA* của anh ta.

Với lược đồ này, miền xác định $\mathcal{D}$ là tập hợp $\mathbf{Z}_n$. Phép biến đổi của một thông điệp M kết hợp với một khóa công $P = (e, n)$ là

$$P(M) = M^e \pmod{n}\ . \qquad (33.39)$$

Phép biến đổi của một văn bản mật mã C kết hợp với một khóa bí mật $S = (d, n)$ là

$$S(C) = C^d \pmod{n}\ . \qquad (33.40)$$

Các phương trình này áp dụng cho cả mật mã hóa lẫn chữ ký. Để tạo một chữ ký, người ký áp dụng khóa bí mật của mình cho thông điệp được ký, thay vì cho một văn bản mật mã. Để xác minh một chữ ký, khóa công của người ký được áp dụng cho nó, thay vì cho một thông điệp được mã hóa.

Để thực thi các phép toán khóa công và khóa bí mật, ta có thể dùng thủ tục MODULAR-EXPONENTIATION mô tả trong Đoạn 33.6. Để phân tích thời gian thực hiện của các phép toán này, ta mặc nhận khóa công (e, n) và khóa bí mật (d, n) thỏa $|e| = O(1)$, $|d| = |n| = \beta$. Như vậy, việc áp dụng một khóa công yêu cầu $O(1)$ phép nhân modula và sử dụng $O(\beta^2)$ phép toán bit. Việc áp dụng một khóa bí mật yêu cầu $O(\beta)$ phép nhân modula, dùng $O(\beta^3)$ phép toán bit.

***Định lý* 33.36 (*Tính đúng đắn của RSA*)**

Các phương trình RSA (33.39) và (33.40) định nghĩa các biến đổi nghịch đảo $\mathbf{Z}_n$ thỏa các phương trình (33.37) và (33.38).

Chứng minh Từ các phương trình (33.39) và (33.40), ta có với bất kỳ $M \in \mathbf{Z}_n$,
$P(S(M)) = S(P(M)) = M^{ed} \pmod{n}$.

Bởi e và d là các nghịch đảo nhân modulo $\phi(n) = (p-1)(q-1)$,
$ed = 1 + k(p-1)(q-1)$

với một số nguyên k. Nhưng như vậy, nếu $M \not\equiv 0 \pmod{p}$, ta có

(dùng Định lý 33.31)

$$\begin{aligned} M^{ed} &\equiv M(M^{p-1})^{k(q-1)} && \pmod p \\ &\equiv M(1)^{k(q-1)} && \pmod p \\ &\equiv M && \pmod p . \end{aligned}$$

Ngoài ra, $M^{ed} \equiv M \pmod p$ nếu $M \equiv 0 \pmod p$. Như vậy,

$M^{ed} \equiv M \pmod p$

với tất cả M. Cũng vậy,

$M^{ed} \equiv M \pmod q$

với tất cả M. Như vậy, theo Hệ luận 33.29 đối với định lý phần dư Trung Quốc,

$M^{ed} \equiv M \pmod n$

với tất cả M.

Tính bảo mật của hệ mật mã RSA phần lớn cậy vào mức bậc khó khăn của việc lấy thừa số các số nguyên lớn. Nếu một kẻ thù có thể lấy thừa số modulus n trong một khóa công, thì hắn có thể suy ra khóa bí mật từ khóa công, dùng kiến thức để các thừa số p và q giống như người tạo ra khóa công đã dùng chúng. Do đó nếu lấy thừa số các số nguyên lớn là dễ dàng, thì việc bẻ hệ mật mã RSA cũng dễ. Đảo đề, cho rằng nếu lấy thừa số các số nguyên lớn khó, thì việc bẻ khóa RSA cũng khó, không được chứng minh. Tuy nhiên, sau một thập kỷ nghiên cứu, người ta chưa tìm thấy phương pháp nào dễ hơn để bẻ hệ mật mã khóa công RSA hơn là lấy thừa số modulus n. Và như sẽ thấy trong Đoạn 33.9, việc lấy thừa số các số nguyên lớn là khó một cách đáng nể. Bằng cách lựa chọn ngẫu nhiên và nhân hai số nguyên tố100-chữ số với nhau, ta có thể tạo một khóa công không thể "bẻ" trong mọi thời lượng khả thi với công nghệ hiện hành. Với sự thiếu vắng của một bước bậct phá căn bản về thiết kế các thuật toán lý thuyết số, hệ mật mã RSA có thể cung cấp một bậc bảo mật cao trong các ứng dụng.

Tuy nhiên, để đạt được tính bảo mật bằng hệ mật mã RSA, ta cần làm việc với các số nguyên có chiều dài 100-200 chữ số, bởi việc lấy thừa số các số nguyên nhỏ hơn là không thực tiễn. Nói cụ thể, ta phải có thể tìm ra các số nguyên tố lớn một cách hiệu quả, để tạo các khóa có chiều dài cần thiết. Bài toán này được đề cập trong Đoạn 33.8.

Vì tính hiệu quả, RSA thường được dùng trong một chế bậc "lai" hoặc "quản lý khóa" với các hệ mật mã hóa phi khóa công nhanh. Với một hệ thống như vậy, các khóa mã hóa và giải mã là như nhau. Nếu Alice muốn gửi riêng một thông điệp dài M cho Bob, cô lựa một khóa

ngẫu nhiên K cho hệ mật mã phi khóa công nhanh và mã hóa M bằng K, đồng thời có được văn bản mật mã C. Ở đây, C dài giống như M, nhưng K thì khá ngắn. Sau đó, cô mã hóa K dùng khóa RSA công của Bob. Bởi K ngắn, nên $P_B(K)$ được tính toán nhanh (nhanh hơn nhiều so với tính toán $P_B(M)$). Sau đó, cô truyền $(C, P_B(K))$ cho Bob, đến lượt Bob sẽ giải mã $P_B(K)$ để có được K rồi sử dụng K để giải mã C, để có được M.

Một cách tiếp cận lai tạo tương tự thường được dùng để tạo các ký danh số hóa một cách hiệu quả. Trong cách tiếp cận này, RSA được tổ hợp với một ***hàm ánh số một chiều*** [one-way hash function] công h— một hàm dễ tính toán nhưng mà về mặt tính toán việc tìm hai thông điệp M và M' là không khả thi sao cho $h(M) = h(M')$. Giá trị $h(M)$ là một "dấu điềm chỉ" ngắn (giả sử, 128 bit) của thông điệp M. Nếu Alice muốn ký nhận một thông điệp M, trước tiên cô áp dụng h cho M để có được dấu điềm chỉ $h(M)$, mà sau đó cô ký bằng khóa bí mật của mình. Cô gửi $(M, S_A(h(M)))$ cho Bob dưới dạng phiên bản M đã ký của cô. Bob có thể xác minh chữ ký bằng cách tính toán $h(M)$ và xác minh P_A được áp dụng cho $S_A(h(M))$ khi nhận sẽ bằng $h(M)$. Bởi không ai có thể tạo hai thông điệp có cùng dấu điềm chỉ, nên không thể thay đổi một thông điệp đã ký và bảo toàn tính hợp lệ của chữ ký.

Cuối cùng, ta lưu ý công dụng của các ***chứng chỉ*** [certificates] khiến việc phân phối các khóa công trở nên dễ dàng hơn nhiều. Ví dụ, mặc nhận có một "cấp thẩm quyền đáng tin cậy" T có khóa công mà mọi người đều biết. Alice có thể lấy từ T một thông điệp đã ký (chứng chỉ của cô) cho biết "khóa công của Alice là P_A." Chứng chỉ này "tự thẩm định quyền" bởi mọi người biết P_T. Alice có thể gộp chứng chỉ của cô với các thông điệp đã ký của mình, sao cho bên nhận có sẵn ngay khóa công của Alice để xác minh chữ ký của cô. Bởi khóa của cô đã được T ký nhận, nên bên nhận biết khóa của Alice đích thực là của Alice.

Bài tập

33.7-1

Xét một khóa RSA được ấn định với $p = 11$, $q = 29$, $n = 319$, và $e = 3$. Giá trị nào của d sẽ được dùng trong khóa bí mật? Nêu việc mật mã hóa thông điệp $M = 100$?

33.7-2

Chứng minh nếu số mũ công của Alice e là 3 và một kẻ thù có được số mũ bí mật của Alice d, thì đối phương có thể lấy thừa số modulus n của Alice trong thời gian đa thức theo số lượng bit trong n. (Mặc dù bạn không được yêu cầu chứng minh nó, song bạn có thể quan tâm muốn

biết rằng kết quả này vẫn đúng cho dù điều kiện $e = 3$ được gỡ bỏ. Xem Miller [147].)

33.7-3 *

Chứng minh RSA là nhân lên theo nghĩa là

$$P_A(M_1)P_A(M_2) \equiv P_A(M_1M_2) \pmod{n}.$$

Dùng sự việc này để chứng minh nếu một kẻ thù có một thủ tục có thể giải mã hiệu quả 1 phần trăm các thông điệp một cách ngẫu nhiên được chọn từ $\mathbf{Z}_n$ và đã mã hóa bằng P_A, thì họ có thể sử dụng một thuật toán theo xác suất để giải mã mọi thông điệp đã mã hóa bằng P_A với xác suất cao.

33.8 Thử tính nguyên

Trong đoạn này, ta xét bài toán tìm các số nguyên tố lớn. Ta bắt đầu bằng phần đề cập đến mật bậc của các số nguyên tố, tiến hành xét một cách tiếp cận hợp lý (nhưng không hoàn chỉnh) đối với phép thử tính nguyên, rồi trình bày một phép thử tính nguyên ngẫu nhiên hóa hiệu quả được xem là của Miller và Rabin.

Mật bậc của các số nguyên tố

Với nhiều ứng dụng (như mật mã hóa), ta cần tìm ra các số nguyên tố "ngẫu nhiên" lớn. May thay, các số nguyên tố lớn không phải là quá hiếm, sao cho không quá tốn thời gian để kiểm tra các số nguyên ngẫu nhiên có kích cỡ thích hợp cho đến khi tìm thấy một số nguyên tố. ***Hàm phân phối số nguyên tố*** $\pi(n)$ chỉ định số lượng các số nguyên tố nhỏ hơn hoặc bằng với n. Ví dụ, $\pi(10) = 4$, bởi có 4 số nguyên tố nhỏ hơn hoặc bằng với 10, tức là, 2, 3, 5, và 7. Định lý số nguyên tố cung cấp một phép xấp xỉ hữu ích cho $\pi(n)$.

Định lý 33.37 (Định lý số nguyên tố)

$$\lim_{n \to \infty} \frac{\pi(n)}{n / \ln n} = 1.$$

Phép xấp xỉ $n/\ln n$ cho ra các ước lượng chính xác hợp lý của $\pi(n)$ thậm chí với n nhỏ. Ví dụ, nó tắt vào khoảng nhỏ hơn 6% tại $n = 10^9$, ở đó $\pi(n) = 50{,}847{,}478$ và $n/\ln n = 48{,}254{,}942$. (Theo một nhà lý thuyết số, 10^9 là một số nhỏ.)

Ta có thể dùng định lý số nguyên tố để ước lượng xác suất mà một số nguyên đã được chọn ngẫu nhiên n sẽ thành ra số nguyên tố dưới dạng $1 / \ln n$. Như vậy, ta cần xem xét xấp xỉ $\ln n$ số nguyên được chọn

một cách ngẫu nhiên gần n để tìm ra một số nguyên tố có cùng chiều dài như n. Ví dụ, để tìm một số nguyên tố 100 chữ số có thể yêu cầu thử xấp xỉ $\ln 10^{100} \approx 230$ con số 100 chữ số được chọn ngẫu nhiên cho tính nguyên. (Con số này có thể được cắt đôi bằng cách chỉ chọn các số nguyên lẻ.)

Trong phần còn lại của đoạn này, ta xét bài toán xác định một số nguyên lẻ lớn n có phải là số nguyên tố hay không. Để tiện dụng về mặt ký hiệu, ta mặc nhận rằng n có phép thừa số hóa số nguyên tố

$$n = p_1^{e_1} p_2^{e_2} \dots p_r^{e_r} , \qquad (33.41)$$

ở đó $r \geq 1$ và $p_1, p_2, \dots, p_r$ là các thừa số nguyên tố của n. Tất nhiên, n là số nguyên tố nếu và chỉ nếu $r = 1$ và $e_1 = 1$.

Một cách tiếp cận đơn giản cho bài toán kiểm tra tính nguyên đó là ***phép chia thử***. Ta thử chia n với mỗi số nguyên 2, 3,..., $\lfloor\sqrt{n}\rfloor$. (Một lần nữa, các số nguyên chẵn lớn hơn 2 có thể bỏ qua.) Ta dễ dàng thấy rằng n là số nguyên tố nếu và chỉ nếu không có ước số thử nào chia hết n. Giả sử mỗi phép chia thử bỏ ra một thời gian bất biến, thời gian thực hiện trường hợp xấu nhất là $\Theta(\sqrt{n})$, là hàm số mũ theo chiều dài của n. (Hãy nhớ lại nếu n được mã hóa theo nhị phân dùng β bit, thì $\beta = \lceil \lg(n + 1) \rceil$, và do đó $\sqrt{n} = \Theta(2^{\beta/2})$.) Như vậy, phép chia thử chỉ làm việc tốt nếu n rất nhỏ hoặc tình cờ có một thừa số nguyên tố nhỏ. Khi làm việc, phép chia thử có ưu điểm đó là nó không những xác định n số nguyên tố hay hợp số mà còn thực tế xác định phép thừa số hóa số nguyên tố nếu n là hợp số.

Trong đoạn này, ta chỉ quan tâm đến việc tìm xem một số đã cho n có phải là số nguyên tố hay không; nếu n là hợp số, ta không quan tâm đến việc tìm phép thừa số hóa số nguyên tố của nó. Như sẽ thấy trong Đoạn 33.9, việc tính toán phép thừa số hóa số nguyên tố của một số quả tốn kém về mặt tính toán. Có lẽ điều đáng ngạc nhiên đó là việc báo một số đã cho có phải là số nguyên tố hay không thường dễ hơn nhiều so với việc xác định phép thừa số hóa số nguyên tố của số đó nếu nó không phải là số nguyên tố.

Phép thử tính nguyên giả

Giờ đây ta xét một phương pháp cho phép thử tính nguyên "hầu như làm việc" và thực tế thích hợp cho nhiều ứng dụng thực tiễn. Phần sau sẽ trình bày cách tu chỉnh phương pháp này để gỡ bỏ khiếm khuyết nhỏ. Cho $\mathbf{Z}^+_n$ thể hiện các thành phần phi zero của $\mathbf{Z}_n$:

$$\mathbf{Z}^+_n = \{1, 2, \dots, n - 1\} .$$

Nếu n là số nguyên tố, thì $\mathbf{Z}^+_n = \mathbf{Z}^*_n$.

Ta nói rằng n là một ***số nguyên tố giả cơ số a*** nếu n là hợp số và

$$a^{n-1} \equiv 1 \pmod{n} . \qquad (33.42)$$

Định lý Fermat (Định lý 33.31) hàm ý rằng nếu n là số nguyên tố, thì n thỏa phương trình (33.42) với mọi a trong $\mathbf{Z}^{+}_{n}$. Như vậy, nếu ta có thể tìm ra bất kỳ $a \in \mathbf{Z}^{+}_{n}$ sao cho n không thỏa phương trình (33.42), thì n chắc chắn là hợp số. Đáng ngạc nhiên, đảo đề hầu như đứng vững, sao cho quy chuẩn này hình thành một đợt trắc nghiệm hầu như hoàn hảo về tính nguyên. Ta trắc nghiệm để xem n thỏa phương trình (33.42) với $a = 2$ hay không. Nếu không, ta khai báo n là hợp số. Bằng không, ta kết xuất một suy đoán rằng n là số nguyên tố (khi, thực tế, tất cả những gì chúng ta biết đó là n là số nguyên tố hoặc một số nguyên tố giả cơ số 2).

Thủ tục dưới đây yêu cầu kiểm tra tính nguyên của n theo cách này. Nó sử dụng thủ tục MODULAR-EXPONENTIATION trong Đoạn 33.6. Đầu vào n được mặc nhận là một số nguyên lớn hơn 2.

PSEUDOPRIME(n)

1 **if** MODULAR-EXPONENTIATION(2, n - 1, n) $\not\equiv 1 \pmod{n}$

2   **then return** COMPOSITE   ▷ Dứt khoát.

3   **else return** PRIME   ▷ Ta hy vọng!

Thủ tục này có thể phạm các lỗi, nhưng chỉ một kiểu. Nghĩa là, nếu nó nói n là hợp số, thì nó luôn đúng. Tuy nhiên, nếu nó nói n là số nguyên tố, thì nó chỉ phạm một lỗi nếu n là một số nguyên tố giả cơ số 2.

Thủ tục này có thường gây lỗi hay không? Thật đáng ngạc nhiên là hiếm khi. Có chỉ 22 giá trị của n nhỏ hơn 10,000 gây lỗi; bốn giá trị đầu tiên như vậy là 341, 561, 645, và 1105. Có thể chứng tỏ rằng xác suất mà chương trình này phạm một lỗi trên một số β-bit được chọn ngẫu nhiên sẽ là zero khi $\beta \to \infty$. Dùng các ước lượng chính xác hơn của Pomerance [157] về số lượng các số nguyên tố giả cơ số 2 có một kích cỡ đã cho, ta có thể ước lượng rằng một số 50 chữ số được chọn ngẫu nhiên được gọi là số nguyên tố theo thủ tục trên đây sẽ có cơ may nhỏ hơn một phần triệu là một số nguyên tố giả cơ số 2, và một số 100 chữ số được chọn ngẫu nhiên được gọi là số nguyên tố sẽ có cơ may nhỏ hơn một trong số 10^{13} là một số nguyên tố giả cơ số 2.

Đáng tiếc, ta không thể loại bỏ tất cả các lỗi bằng cách đơn giản kiểm tra phương trình (33.42) để tìm một số cơ cố hai, giả sử $a = 3$, bởi có các hợp số nguyên n thỏa phương trình (33.42) với *tất cả* $a \in \mathbf{Z}^{*}_{n}$. Các số nguyên này được xem là ***các con số Carmichael.*** Ba con số

Carmichael đầu tiên là 561, 1105, và 1729. Các con số Carmichael cực kỳ hiếm; ví dụ, chỉ có 255 trong số chúng nhỏ hơn 100,000,000. Bài tập 33.8-2 sẽ giúp giải thích tại sao chúng lại hiếm.

Kế tiếp, ta nêu cách cải thiện phép thử tính nguyên của chúng ta sao cho nó không bị các con số Carmichael đánh lừa.

Phép thử tính nguyên ngẫu nhiên hóa Miller-Rabin

• Phép thử tính nguyên Miller-Rabin khắc phục các bài toán của trắc nghiệm PSEUDOPRIME đơn giản bằng hai chi tiết sửa đổi:

• Nó thử vài các giá trị cơ số được chọn ngẫu nhiên a thay vì chỉ một giá trị cơ số. Trong khi tính toán mỗi mũ hóa modula, nó lưu ý xem có phát hiện một căn bậc hai không hiển nhiên của 1, modulo n, không. Nếu có, nó dừng và kết xuất COMPOSITE. Hệ luận 33.35 chứng minh sự phát hiện các hợp số theo cách này.

Dưới đây là mã giả của phép thử tính nguyên Miller-Rabin. Đầu vào $n > 2$ là số lẻ được trắc nghiệm về tính nguyên, và s là số lượng các giá trị cơ số được chọn ngẫu nhiên từ $\mathbf{Z}^+_n$ để thử. Mã sử dụng bộ sinh ngẫu số RANDOM trong Đoạn 8.3: RANDOM(1, $n-1$) trả về một số nguyên a được chọn ngẫu nhiên thỏa $1 \leq a \leq n-1$. Mã sử dụng một thủ tục phụ WITNESS sao cho WITNESS(a, n) là TRUE nếu và chỉ nếu a là một "bằng chứng" về tính hợp số của n—nghĩa là, nếu nó có thể dùng a để chứng minh (theo cách mà ta sẽ thấy) rằng n là hợp số. Trắc nghiệm WITNESS(a, n) cũng tương tự như, song hiệu quả hơn, trắc nghiệm

$$a^{n-1} \not\equiv 1 \pmod n$$

đã lập thành cơ sở (dùng $a = 2$) cho PSEUDOPRIME. Trước tiên ta trình bày và xác minh phấn kiến tạo WITNESS, rồi nêu cách dùng nó trong phép thử tính nguyên Miller-Rabin.

```
WITNESS(a, n)
cho <b_k, b_{k-1}, ..., b_0> là phần biểu diễn nhị phân của n − 1
d ← 1
for i ← k downto 0
    do x ← d
       d ← (d • d) mod n
       if d = 1 and x ≠ 1 and x ≠ n − 1
          then return TRUE
       if b_i = 1
```

```
9                    then d ← (d • a) mod n
10 if d ≠ 1
11       then return TRUE
12 return FALSE
```

Mã giả này của WITNESS dựa trên mã giả của thủ tục MODULAR-EXPONENTIATION. Dòng 1 xác định phần biểu diễn nhị phân của n - 1, sẽ được dùng để nâng a thành lũy thừa thứ (n – 1). Các dòng 3-9 tính toán d dưới dạng a^{n-1} mod n. Phương pháp được dùng cũng giống như trong MODULAR-EXPONENTIATION. Tuy nhiên, mỗi khi thực hiện một bước bình phương trên dòng 5, các dòng 6-7 kiểm tra xem đã khám phá một căn bậc hai không hiển nhiên của 1 hay chưa. Nếu có, thuật toán dừng và trả về TRUE. Các dòng 10-11 trả về TRUE nếu giá trị đã tính toán với a^{n-1} mod n không bằng 1, giống như thủ tục PSEUDOPRIME trả về COMPOSITE trong trường hợp này.

Giờ đây ta chứng minh rằng nếu WITNESS(a, n) trả về TRUE, thì có thể dùng a kiến tạo một phần chứng minh rằng n là hợp số.

Nếu WITNESS trả về TRUE từ dòng 11, thì nó đã khám phá ra $d = a^{n-1}$ mod $n \neq 1$. Tuy nhiên, nếu n là số nguyên tố, thì theo định lý Fermat (Định lý 33.31) ta có $a^{n-1} \equiv 1 \pmod{n}$ với tất cả $a \in \mathbf{Z}^+_n$. Do đó, n không thể là số nguyên tố, và phương trình a^{n-1} mod $1 \neq 1$ là một chứng minh cho sự việc này.

Nếu WITNESS trả về TRUE từ dòng 7, thì nó đã khám phá ra x là một căn bậc hai không hiển nhiên của 1, modulo n, bởi ta có $x \not\equiv \pm 1 \pmod{n}$ yet $x^2 \equiv 1 \pmod{n}$. Hệ luận 33.35 phát biểu rằng chỉ nếu n là hợp số ta có thể có một căn bậc hai không hiển nhiên của 1 modulo n, để phần biểu hiện x là một căn bậc hai không hiển nhiên của 1 modulo n sẽ là phần chứng minh rằng n là hợp số.

Điều này hoàn tất phần chứng minh của chúng ta về tính đúng đắn của WITNESS. Nếu lần triệu gọi WITNESS(a, n) kết xuất TRUE, thì n chắc chắn là hợp số, và phần chứng minh rằng n là hợp số có thể dễ dàng xác định từ a và n. Giờ đây ta xét phép thử tính nguyên Miller-Rabin dựa trên công dụng của WITNESS.

```
MILLER-RABIN(n, s)
for j ← 1 to s
    do a ← RANDOM(1, n – 1)
       if WITNESS(a, n)
          then return COMPOSITE          ▷ Dứt khoát.
```

5 **return** PRIME ▷ Hầu như chắc chắn.

Thủ tục MILLER-RABIN là một tìm kiếm theo xác suất một dạng chứng minh rằng n là hợp số. Vòng lặp chính (bắt đầu trên dòng 1) chọn s giá trị ngẫu nhiên của a từ $\mathbf{Z}^+_n$ (dòng 2). Nếu một trong số các a đã chọn là một bằng chứng cho tính hợp số n, thì MILLER-RABIN kết xuất COMPOSITE trên dòng 4. Một kết xuất như vậy luôn đúng đắn, theo tính đúng đắn của WITNESS. Nếu không tìm thấy bằng chứng nào trong s lần thử, MILLER-RABIN mặc nhận là vậy bởi vì không có bằng chứng nào được tìm thấy, và do đó n là số nguyên tố. Ta sẽ thấy kết xuất này ắt là đúng đắn nếu s đủ lớn, nhưng có một cơ may nhỏ rằng thủ tục có thể không may trong khi chọn lựa a và các bằng chứng vẫn tồn tại cho dù không tìm thấy chúng.

Để minh họa phép toán của MILLER-RABIN, cho n là số Carmichael 561. Giả sử $a = 7$ được chọn làm cơ số, Hình 33.4 cho thấy WITNESS phát hiện một căn bậc hai không hiển nhiên của 1 trong bước bình phương cuối, bởi $a^{280} \equiv 67 \pmod{n}$ và $a^{560} \equiv 1 \pmod{n}$. Do đó, $a = 7$ là một bằng chứng cho tính hợp số của n, WITNESS(7, n) trả về TRUE, và MILLER-RABIN trả về COMPOSITE.

Nếu n là một số β-bit, MILLER-RABIN yêu cầu $O(s\beta)$ phép toán số học và $O(s\beta^3)$ phép toán bit, bởi nó yêu cầu theo tiệm cận không làm việc hơn mức s mũ hóa modula.

Tỷ lệ lỗi của phép thử tính nguyên Miller-Rabin

Nếu MILLER-RABIN kết xuất PRIME, thì có một cơ may nhỏ rằng nó đã phạm một lỗi. Tuy nhiên, khác với PSEUDOPRIME, cơ may lỗi không tùy thuộc vào n; không có đầu vào tồi cho thủ tục này. Thay vì thế, nó tùy thuộc vào kích cỡ của s và "số phận" trong khi chọn các giá trị cơ số a. Ngoài ra, do mỗi lần trắc nghiệm thường nghiêm ngặt hơn so với một đợt kiểm tra đơn giản của phương trình (33.42), nên theo nguyên tắc chung ta có thể dự kiến tỷ lệ lỗi là nhỏ đối với các số nguyên n được chọn ngẫu nhiên. Định lý dưới đây trình bày một lập luận chính xác hơn.

Định lý 33.38

Nếu n là một hợp số lẻ, thì số lượng các bằng chứng đối với tính hợp số của n ít nhất là $(n-1)/2$.

Chứng minh Phần chứng minh chứng tỏ số lượng không bằng chứng không nhiều hơn $(n-1)/2$, hàm ý định lý.

Trước tiên, ta nhận thấy mọi phi bằng chứng phải là một phần tử của $\mathbf{Z}^*_n$, bởi mọi phi bằng chứng a thỏa $a^{n-1} \equiv 1 \pmod{n}$, hơn nữa nếu gcd($a$,

$n) = d > 1$, thì không có nghiệm x cho phương trình $ax \equiv 1 \pmod{n}$, theo Hệ luận 33.21. (Nói cụ thể, $x = a^{n-2}$ không phải là một nghiệm.) Như vậy mọi phần tử của $\mathbf{Z}_n - \mathbf{Z}^*_n$ là một bằng chứng cho tính hợp số của n.

Để hoàn tất phần chứng minh, ta chứng tỏ tất cả các phản chứng được chứa trong một nhóm con riêng B của $\mathbf{Z}^*_n$. Theo Hệ luận 33.16, như vậy ta có $|B| \leq |\mathbf{Z}^*_n|/2$. Bởi $|\mathbf{Z}^*_n| \leq n - 1$, ta được $|B| \leq (n-1)/2$. Do đó, số lượng các phi bằng chứng tối đa là $(n-1)/2$, sao cho số lượng các bằng chứng phải ít nhất là $(n-1)/2$.

Giờ đây, ta nêu cách tìm một nhóm con riêng B của $\mathbf{Z}^*_n$ chứa tất cả các phi bằng chứng. Ta tách phần chứng minh thành hai trường hợp.

Trường hợp 1: Ở đó tồn tại $x \in \mathbf{Z}^*_n$ sao cho

$$x^{n-1} \not\equiv 1 \pmod{n}. \qquad (33.43)$$

Cho $B = \{b \in \mathbf{Z}^*_n : b^{n-1} \equiv 1 \pmod{n}\}$. Bởi B được đóng dưới phép nhân modulo n, ta có B là một nhóm con của $\mathbf{Z}^*_n$ theo Định lý 33.14. Lưu ý, mọi phi bằng chứng đều thuộc về B, bởi một phi bằng chứng a thỏa $a^{n-1} \equiv 1 \pmod{n}$. Do $x \in \mathbf{Z}^*_n - B$, ta có B là một nhóm con riêng của $\mathbf{Z}^*_n$.

Trường hợp 2: Với tất cả $x \in \mathbf{Z}^*_n$,

$$x^{n-1} \not\equiv 1 \pmod{n}. \qquad (33.44)$$

Trong trường hợp này, n không thể là một lũy thừa số nguyên tố. Để xem tại sao, ta cho $n = p^e$, ở đó p là một số nguyên tố lẻ và $e > 1$. Định lý 33.32 hàm ý rằng $\mathbf{Z}^*_n$ chứa một thành phần g sao cho $\text{ord}_n(g) = |\mathbf{Z}^*_n| = \phi(n) = (p-1)p^{e-1}$. Nhưng như vậy phương trình (33.44) và định lý lôga rời rạc (Định lý 33.33, lấy $y = 0$) hàm ý rằng $n - 1 \equiv 0 \pmod{\phi(n)}$, hoặc

$$(p-1)p^{e-1} \mid p^e - 1.$$

Điều kiện này thất bại với $e > 1$, bởi như vậy bên phía trái chia được cho p nhưng bên phía phải thí không. Do đó, n không phải là một số nguyên tố lũy thừa.

Bởi n không phải là một số nguyên tố lũy thừa, ta phân tích nó thành một tích $n_1 n_2$, ở đó n_1 và n_2 lớn hơn 1 và là nguyên tố cùng nhau. (Có thể có vài cách để thực hiện điều này, và bất kể ta chọn cách nào. Ví dụ, nếu $n = p_1^{e_1} p_2^{e_2} \dots p_r^{e_r}$, thì ta có thể chọn $n_1 = p_1^{e_1}$ và $n_2 = p_2^{e_2} p_3^{e_3} \dots p_r^{e_r}$.)

Định nghĩa t và u sao cho $n - 1 = 2^t u$, ở đó $t \geq 1$ và u là lẻ. Với mọi $a \in \mathbf{Z}^+_n$, ta xét dãy

$$\hat{a} = \langle a^u, a^{2u}, a^{2^2 u} \dots, a^{2^t u} \rangle, \qquad (33.45)$$

ở đó tất cả các thành phần được tính toán modulo n. Bởi $2^t \mid n - 1$, phần biểu diễn nhị phân của $n - 1$ kết thúc trong t zero, và các thành phần của $\hat{a}$ là $t + 1$ giá trị chót của d được WITNESS tính toán trong một

phép tính a^{n-1} mod n; t phép toán chót là các phép bình phương.

Giờ đây ta tìm $j \in \{0, 1, ..., t\}$ sao cho ở đó tồn tại $v \in \mathbf{Z}^*_n$ sao cho $v^{2^j u} \equiv -1 \pmod{n}$; j sẽ càng lớn càng tốt. Một j như vậy chắc chắn tồn tại bởi u là lẻ: ta có thể chọn $v = -1$ và $j = 0$. Chỉnh v để thỏa điều kiện đã cho. Cho

$$B = \{x \in \mathbf{Z}^*_n : x^{2^j u} \equiv \pm 1 \pmod{n}\}.$$

Bởi B được đóng dưới phép nhân modulo n, nó là một nhóm con của $\mathbf{Z}^*_n$. Do đó, $|B|$ chia hết $|\mathbf{Z}^*_n|$. Mọi phi bằng chứng phải là một phần tử của B, bởi dãy (33.45) mà một phi bằng chứng tạo phải là tất cả các 1 nếu không sẽ chứa một -1 không trễ hơn vị trí thứ j, theo tính tột bậc [maximality] của j.

Giờ đây ta dùng sự tồn tại của v để chứng minh ở đó tồn tại một $w \in \mathbf{Z}^*_n - B$. Bởi $v^{2^j u} \equiv -1 \pmod{n}$, ta có $v^{2^j u} \equiv -1 \pmod{n_1}$ theo Hệ luận 33.29. Theo Hệ luận 33.28, có một w đồng thời thỏa các phương trình

$$w \equiv v \pmod{n_1}$$

$$w \equiv 1 \pmod{n_2}.$$

Do đó,

$$w^{2^j u} \equiv -1 \pmod{n_1},$$

$$w^{2^j u} \equiv 1 \pmod{n_2}.$$

Cùng với Hệ luận 33.29, các phương trình này hàm ý rằng

$$w^{2^j u} \not\equiv \pm 1 \pmod{n}, \qquad (33.46)$$

và do đó $w \notin B$. Bởi $v \in \mathbf{Z}^*_n$, ta có $v \in \mathbf{Z}^*_n$. Như vậy, $w \in \mathbf{Z}^*_n$, và do đó $w \in \mathbf{Z}^*_n - B$. Ta kết luận rằng B là một nhóm con riêng của $\mathbf{Z}^*_n$.

Trong cả hai trường hợp, ta thấy rằng số lượng các bằng chứng về tính hợp số của n ít nhất là $(n-1)/2$.

Định lý 33.39

Với bất kỳ số nguyên lẻ $n > 2$ và các số nguyên dương s, xác suất mà MILLER-RABIN(n, s) gặp lỗi tối đa là 2^{-s}.

Chứng minh

Dùng Định lý 33.38, ta thấy nếu n là hợp số, thì mỗi lần thi hành vòng lặp của các dòng 1-4 đều có một xác suất ít nhất là 1/2 để khám phá một bằng chứng x đối với tính hợp số của n. MILLER-RABIN chỉ phạm một lỗi nếu nó không may mắn bỏ sót việc khám phá một bằng chứng về tính hợp số của n trên mỗi trong số s lần lặp lại của vòng lặp chính. Xác suất của một chuỗi cơ hội bỏ lỡ như vậy tối đa là 2^{-s}.

Như vậy, việc chọn $s = 50$ ắt là đủ cho hầu hết mọi ứng dụng có thể tưởng tượng được. Nếu ta đang gắng tìm các số nguyên tố lớn bằng cách áp dụng MILLER-RABIN cho các số nguyên lớn *được chọn ngẫu nhiên*, thì có thể lập luận (mặc dù ta sẽ không làm vậy ở đây) rằng việc chọn một giá trị nhỏ của s (giả sử 3) ắt hẳn không thể dẫn đến các kết quả lỗi. Nghĩa là, với một hợp số nguyên lẻ n được chọn ngẫu nhiên, số lượng dự kiến các phi bằng chứng đối với tính hợp số của n ắt nhỏ hơn nhiều so với $(n - 1)/2$. Tuy nhiên, nếu số nguyên n không được chọn một cách ngẫu nhiên, khả năng tốt nhất có thể được chứng minh đó là số lượng các phi bằng chứng tối đa là $(n - 1)/4$, dùng một phiên bản đã cải thiện của Định lý 33.39. Vả lại, ở đó tồn tại các số nguyên n mà số lượng các phi bằng chứng là $(n - 1)/4$.

Bài tập

33.8-1

Chứng minh nếu một số nguyên $n > 1$ không phải là một số nguyên tố hoặc một lũy thừa số nguyên tố, thì ở đó tồn tại một căn bậc hai không hiển nhiên của 1 modulo n.

33.8-2 *

Có thể củng cố phần nào định lý Euler theo dạng

$a^{\lambda(n)} \equiv 1 \pmod{n}$ với tất cả $a \in \mathbf{Z}^*_n$,

ở đó $\lambda(n)$ được định nghĩa bởi

$$\lambda(n) = \text{lcm}(\phi(p_1^{e_1}), \ldots, \phi(p_r^{e_r})). \qquad (33.47)$$

Chứng minh $\lambda(n) \mid \phi(n)$. Một hợp số n là một số Carmichael nếu $\lambda(n) \mid n - 1$. Số Carmichael nhỏ nhất là $561 = 3 \cdot 11 \cdot 17$; ở đây, $\lambda(n) = \text{lcm}(2, 10, 16) = 80$, chia hết cho 560. Chứng minh các con số Carmichael phải vừa "không bình phương" (không chia được bởi bình phương của bất kỳ số nguyên tố nào) và tích của ít nhất ba số nguyên tố. Vì lý do này, chúng không rất chung.

33.8-3

Chứng minh nếu x là một căn bậc hai không hiển nhiên của 1, modulo n, thì cả $\gcd(x - 1, n)$ lẫn $\gcd(x + 1, n)$ là những ước số không hiển nhiên của n.

* 33.9 Phép thừa số hóa số nguyên

Giả sử ta có một số nguyên n mà ta muốn ***lấy thừa số***, nghĩa là, phân tích thành một tích các số nguyên tố. Phép thử tính nguyên của đoạn trước sẽ báo ta biết n là hợp số, nhưng nó thường không báo cho ta biết các thừa số nguyên tố của n. Việc lấy thừa số một số nguyên lớn n dường như khó hơn nhiều so với việc đơn giản xác định n là số nguyên tố hay hợp số. Việc thừa số hóa một số 200 chữ số thập phân tùy ý là không khả thi đối với các siêu máy tính ngày nay và các thuật toán tốt nhất hiện nay.

Heuristic rho của Pollard

Phép chia thử với tất cả các số nguyên lên tới B được bảo đảm thừa số hóa hoàn toàn mọi số lên tới B^2. Với cùng lượng công việc, thủ tục dưới đây sẽ thừa số hóa mọi số lên tới B^4 (trừ phi ta không may mắn). Bởi thủ tục chỉ là một heuristic, nên thời gian thực hiện cũng như sự thành công của nó đều không được bảo đảm, mặc dù thủ tục rất hiệu quả trong thực tế.

```
POLLARD-RHO(n)
i ← 1
x_1 ← RANDOM(0, n − 1)
y ← x_1
k ← 2
while TRUE
      do i ← i + 1
         x_i ← (x_{i-1}^2 − 1) mod n
         d ← gcd(y − x_i, n)
         if d ≠ 1 và d ≠ n
            then print d
         if i = k
            then y ← x_i
                 k ← 2k
```

Thủ tục làm việc như sau. Các dòng 1-2 khởi tạo i theo 1 và x_1 theo một giá trị được chọn ngẫu nhiên trong $\mathbf{Z}_n$. Vòng lặp **while** bắt đầu trên dòng 5 lặp lại hoài, tìm kiếm các thừa số của n. Trong mỗi lần lặp lại của vòng lặp **while**, phép truy toán

$$x_i \leftarrow (x_{i-1}) \bmod n \tag{33.48}$$

được dùng trên dòng 7 để tạo giá trị kế tiếp của x_i trong dãy vô hạn

$$x_1,\ x_2,\ x_3,\ x_4,\ \ldots\ ; \tag{33.49}$$

giá trị của i được gia số tương ứng trên dòng 6. Để minh bạch, mã được viết dùng các biến x_i được viết dưới, nhưng chương trình làm việc y hệt nếu loại tất cả các con chữ dưới, bởi chỉ cần duy trì giá trị mới nhất của x_i.

Thỉnh thoảng, chương trình lưu giá trị x_i mới phát sinh trong biến y. Cụ thể, các giá trị được lưu là những có các con chữ dưới là các lũy thừa của 2:

$$x_1, x_2, x_4, x_8, x_{16}, \ldots .$$

Dòng 3 lưu giá trị x_i, và dòng 12 lưu x_k mỗi khi i bằng k. Biến k được khởi tạo theo 2 trong dòng 4, và k được nhân đôi trong dòng 13 mỗi khi y được cập nhật. Do đó, k theo dãy 1, 2, 4, 8,...và luôn cho con chữ dưới của giá trị kế tiếp x_k được lưu trong y.

Các dòng 8-10 gắng tìm một thừa số của n, dùng giá trị đã lưu của y và giá trị hiện hành của x_i. Cụ thể, dòng 8 tính ước số chung lớn nhất $d = \gcd(y - x_i, n)$. Nếu d là một ước số không hiển nhiên của n (được kiểm tra trong dòng 9), thì dòng 10 sẽ in d.

Thoạt đầu thủ tục để tìm một thừa số này có vẻ hơi khó hiểu. Tuy nhiên, lưu ý POLLARD-RHO không bao giờ in một đáp án sai; mọi số mà nó in là một ước số không hiển nhiên của n. Tuy vậy, POLLARD-RHO có thể không in gì cả; không có gì bảo đảm rằng nó sẽ tạo ra một kết quả bất kỳ. Tuy nhiên, như sẽ thấy, ta có thể an tâm dự kiến POLLARD-RHO sẽ in một thừa số p của n sau xấp xỉ $\sqrt{p}$ lần lặp lại của vòng lặp **while.** Như vậy, nếu n là hợp số, ta có thể dự kiến thủ tục này phát hiện đủ ước số để thừa số hóa n hoàn toàn sau xấp xỉ $n^{1/4}$ cập nhật, bởi mọi thừa số nguyên tố p của n ngoại trừ có thể giá trị lớn nhất nhỏ hơn $\sqrt{n}$.

Ta phân tích cách ứng xử của thủ tục này bằng cách nghiên cứu thời lượng của một dãy ngẫu nhiên modulo n lặp lại một giá trị. Bởi $\mathbf{Z}_n$ là hữu hạn, và bởi mỗi giá trị trong dãy (33.49) chỉ tùy thuộc vào giá trị trước đó, nên dãy (33.49) chung cuộc tự lặp lại. Một khi ta đụng một x_i sao cho $x_i = x_j$ với một $j < i$, ta nằm trong một chu trình, bởi $x_{i+1} = x_{j+1}$ $x_{i+2} = x_{j+2}$, và vân vân. Sở dĩ có tên là "heuristic rho " đó là, như Hình 3 3. 7 nêu, dãy $x_1, x_2, ..., x_{j-1}$ có thể được vẽ dưới dạng "đuôi" của chữ rho, và chu trình $x_j, x_{j+1}, ..., x_i$ dưới dạng "thân" của rho.

Ta hãy xét câu hỏi dãy của x_i lặp lại trong bao lâu. Điều này không phải chính xác là cái ta cần, nhưng như vậy ta sẽ thấy cách sửa đổi đối số.

Vì sự đánh giá này, ta hãy mặc nhận rằng hàm $(x^2 - 1) \bmod n$ ứng xử như một hàm "ngẫu nhiên." Tất nhiên, nó thực sự không phải là ngẫu nhiên, nhưng giả thiết này cho ra các kết quả nhất quán với cách ứng xử đã nhận xét của POLLARD-RHO. Như vậy, ta có thể xét mỗi x_i đã được rút độc lập từ $\mathbf{Z}_n$ theo một phép phân phối đều trên $\mathbf{Z}_n$. Theo sự phân tích nghịch lý ngày sinh nhật trong Đoạn 6.6.1, số lượng các bước dự trù diễn ra trước dãy các chu trình là $\Theta(\sqrt{n})$.

Giờ đây với sự sửa đổi cần thiết. Cho p là một thừa số không hiển nhiên của n sao cho $\gcd(p, n/p) = 1$. Ví dụ, nếu n có phép thừa số hóa $n = p_1^{e_1}, p_2^{e_2} ... p_r^{e_r}$, thì ta có thể lấy p là $p_1^{e_1}$. (Nếu $e_1 = 1$, thì p chính là thừa số nguyên tố nhỏ nhất của n, một ví dụ hay cần nhớ kỹ.) Dãy $<x_i>$ cảm sinh một dãy tương ứng $<x'_i>$ modulo p, ở đó

$x'_i = x''_i \bmod p$

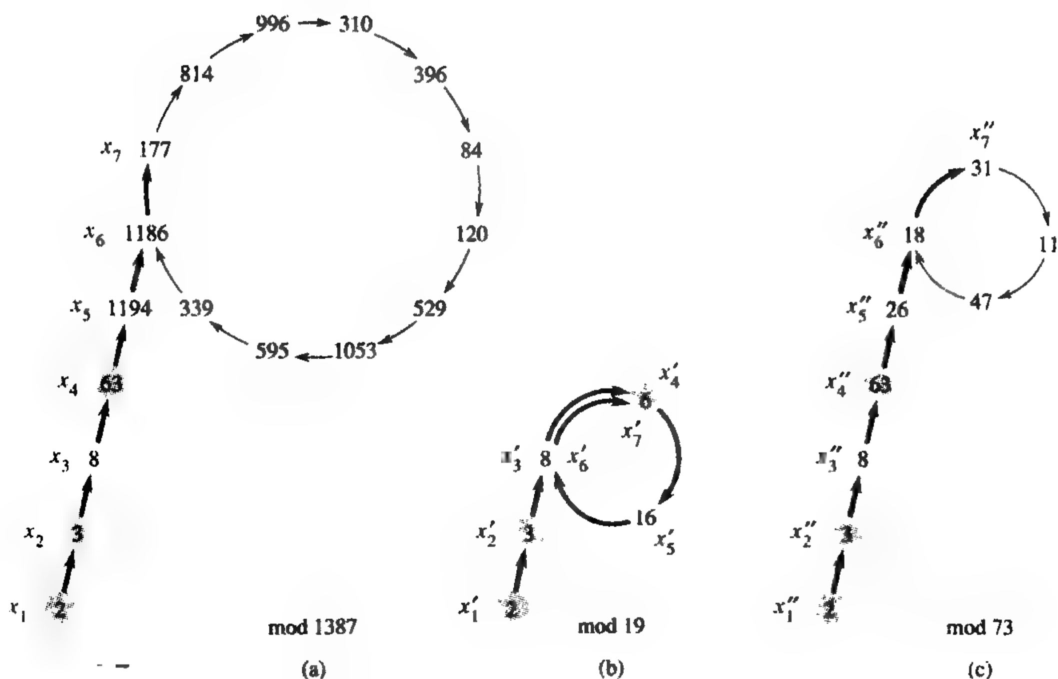
với tất cả i. Vả lại, do định lý phần dư Trung Quốc ta có

$$x'_{i+1} = (x'^2_i - 1) \bmod p \tag{33.50}$$

bởi

$(x \bmod n) \bmod p = x \bmod p$,

theo Bài tập 33.1-6.



Hình 33.7 Heuristic rho của Pollard. **(a)** Các giá trị được tạo bằng phép truy toán $x_{i+1} \leftarrow (x_i^2 - 1) \bmod 1387$, bắt đầu với $x_1 = 2$. Phép thừa số hóa số nguyên tố của 1387 là 19 . 73. Các mũi tên đậm nêu rõ các bước lặp lại được thi hành trước khi thừa số 19 được khám phá. Các mũi tên nhạt trỏ đến các giá trị không đụng trong lần lặp lại, để minh họa hình dáng "rho". Các giá trị tô bóng là các giá trị y được lưu trữ bởi POLLARD-RHO. Thừa số 19 được khám phá sau khi đụng $x_7 = 177$, khi gcd(63 - 177, 1387) = 19 được tính toán. Giá trị x_i đầu tiên sẽ được lặp lại là 1186, nhưng thừa số 19 được khám phá trước khi đạt đến giá trị này. **(b)** Các giá trị được tạo bởi cùng phép truy toán, modulo 19. Mọi giá trị x_i đã cho trong phần (a) là tương đương, modulo 19, với giá trị x'_i nêu ở đây. Ví dụ, cả $x_4 = 63$ lẫn $x_7 = 177$ tương đương với 6, modulo 19. **(c)** Các giá trị được tạo bởi cùng phép truy toán, modulo 73. Mọi giá trị x_i đã cho trong phần (a) tương đương, modulo 73, với giá trị x''_i nêu ở đây. Theo định lý phần dư Trung Quốc, mỗi mắt trong phần (a) tương ứng với một cặp các mắt, một từ phần (b) và một từ phần (c).

Biện luận như trước, ta thấy rằng số lượng các bước dự trù trước khi dãy $<x'_i>$ lặp lại là $\Theta(\sqrt{p})$. Nếu p là nhỏ so với n, dãy $<x'_i>$ có thể lặp lại nhanh hơn nhiều so với dãy $<x_i>$. Quả vậy, dãy $<x'_i>$ lặp lại ngay khi hai thành phần của dãy $<x_i>$ đơn thuần tương đương modulo p, thay vì tương đương modulo n. Xem Hình 33.7, các phần (b) và (c), để có phần minh họa.

Cho t thể hiện chỉ số của giá trị được lặp lại đầu tiên trong dãy $<x'_i>$, và cho $u > 0$ thể hiện chiều dài của chu trình đã được tạo bằng cách ấy. Nghĩa là, t và $u > 0$ là các giá trị nhỏ nhất sao cho $x'_{t+i} = x'_{t+u+i}$ với tất cả $i \geq 0$. Theo các lập luận trên đây, cả hai giá trị dự trù của t và u là $\Theta(\sqrt{p})$. Lưu ý, nếu $x'_{t+i} = x'_{t+u+i}$, thì $p \mid (x_{t+u+i} - x_{t+i})$. Như vậy, $\gcd(x_{t+u+i} - x_{t+i}, n) > 1$.

Do đó, một khi POLLARD-RHO đã lưu dưới dạng y bất kỳ giá trị x_k sao cho $k \geq t$, thì $y \bmod p$ luôn nằm trên chu trình modulo p. (Nếu một giá trị mới được lưu dưới dạng y, giá trị đó cũng nằm trên chu trình modulo p.) Cuối cùng, k được ấn định theo một giá trị lớn hơn u, và như vậy thủ tục thực hiện nguyên một vòng lặp quanh chu trình modulo p mà không thay đổi giá trị của y. Như vậy, một thừa số của n được khám phá khi x_i "đụng" giá trị được lưu trữ trước đó của y, modulo p, nghĩa là, khi $x_i \equiv y \pmod{p}$.

Có lẽ, thừa số tìm thấy là thừa số p, mặc dù thỉnh thoảng có thể xảy ra trường hợp một bội của p được khám phá. Bởi các giá trị dự kiến của cả t và u là $\Theta(\sqrt{p})$, số lượng các bước dự trù cần thiết để tạo thừa số p là $\Theta(\sqrt{p})$.

Có hai lý do giải thích tại sao thuật toán này không thể thực hiện đúng như dự kiến. Trước tiên, việc phân tích heuristic của thời gian thực hiện không nghiêm ngặt, và có thể chu trình của các giá trị, modulo p, lớn hơn nhiều so với $\sqrt{p}$. Trong trường hợp này, thuật toán thực hiện đúng đắn nhưng chậm hơn nhiều so với mong muốn. Trong thực tế, điều này dường như không thành vấn đề. Thứ hai, các ước số của n được tạo bởi thuật toán này có thể luôn là một trong các thừa số hiển nhiên 1 hoặc n. Ví dụ, giả sử $n = pq$, ở đó p và q là số nguyên tố. Có thể xảy ra trường hợp các giá trị của t và u cho p đồng nhất với các giá trị của t và u cho q, và như vậy thừa số p luôn được biểu lộ trong cùng phép toán gcd biểu lộ thừa số q. Bởi cả hai thừa số được biểu lộ cùng một lúc, nên thừa số hiển nhiên $pq = n$ được biểu lộ, là vô ích. Một lần nữa, điều này dường như không phải là một bài toán thực trong thực tế. Nếu cần, heuristic có thể được khởi bậcng lại bằng một phép truy toán kháv có dạng $x_{i+1} \leftarrow (x_i^2 - c) \bmod n$. (Các giá trị $c = 0$ và $c = 2$ phải được tránh vì các lý do mà ta không đề cập ở đây, nhưng

các giá trị khác là ổn thỏa.)

Tất nhiên, phân tích này là heuristic và không nghiêm ngặt, bởi phép truy toán không thực tế "ngẫu nhiên." Tuy nhiên, thủ tục thực hiện tốt trong thực tế, và dường như nó cũng hiệu quả như phần phân tích heuristic này nêu rõ. Nó là phương pháp chọn lựa để tìm các thừa số nguyên tố nhỏ của một số lớn. Để thừa số hóa một hợp số β-bit n hoàn toàn, ta chỉ cần tìm ra tất cả các thừa số nguyên tố nhỏ hơn $\lfloor n^{1/2} \rfloor$, và do đó ta dự kiến POLLARD-RHO yêu cầu tối đa $n^{1/4} = 2^{\beta/4}$ phép toán số học và tối đa $n^{1/4}\beta^3 = 2^{\beta/4}\beta^3$ phép toán bit. Khả năng của POLLARD-RHO để tìm một thừa số nhỏ p của n với một số dự kiến $\Theta(\sqrt{p})$ phép toán số học thường là tính năng hấp dẫn nhất của nó.

Bài tập

33.9-1

Tham khảo lịch sử thi hành nêu trong Hình 33.7(a), POLLARD-RHO in thừa số 73 của 1387 khi nào?

33.9-2

Giả sử ta có một hàm $f : \mathbf{Z}_n \to \mathbf{Z}_n$ và một giá trị ban đầu $x_0 \in \mathbf{Z}_n$. Định nghĩa $x_i = f(x_{i-1})$ với $i = 1, 2,\ldots$.Cho t và $u > 0$ là các giá trị nhỏ nhất sao cho $x_{t+i} = x_{t+u+i}$ với $i = 0, 1,\ldots$.Trong thuật ngữ của thuật toán rho của Pollard, t là chiều dài của đuôi và u là chiều dài của chu trình của rho. Nêu một thuật toán hiệu quả để xác định t và u chính xác, và phân tích thời gian thực hiện của nó.

33.9-3

Bạn dự kiến POLLARD-RHO yêu cầu bao nhiêu bước để phát hiện một thừa số có dạng p^e, ở đó p là số nguyên tố và $e > 1$?

33.9-4 *

Một nhược điểm của POLLARD-RHO như đã viết đó là nó yêu cầu một phép tính gcd cho mỗi bước của phép truy toán. Đã có đề nghị rằng ta có thể tạo lô các phép tính gcd bằng cách tích lũy tích của vài x_i trong một hàng rồi lấy gcd của tích này với y đã lưu. Mô tả cẩn thận cách thức mà bạn thực thi ý tưởng này, tại sao nó làm việc, và nêu kích cỡ lô mà bạn sẽ chọn dưới dạng hiệu quả nhất khi làm việc trên một số β-bit n.

Các Bài Toán

33-1 Thuật toán gcd nhị phân

Trên hầu hết các máy tính, các phép toán trừ, trắc nghiệm tính chẵn

lẻ (lẻ hoặc chẵn) của một số nguyên nhị phân, và việc chia đôi có thể được thực hiện nhanh hơn so với việc tính toán các số dư. Bài toán này nghiên cứu ***thuật toán gcd nhị phân,*** tránh các phép tính số dư được dùng trong thuật toán Euclid.

a. Chứng minh nếu cả a và b đều là chẵn, thì $\gcd(a, b) = 2 \gcd(a/2, b/2)$.

b. Chứng minh nếu a là lẻ và b là chẵn, thì $\gcd(a, b) = \gcd(a, b/2)$.

c. Chứng minh nếu cả a và b đều là lẻ, thì $\gcd(a, b) = \gcd((a - b)/2, b)$.

d. Thiết kế một thuật toán gcd nhị phân hiệu quả với các số nguyên đầu vào a và b, ở đó $a \geq b$, chạy trong $O(\lg(\max(a, b)))$ thời gian. Giả sử mỗi phép trừ, kiểm tra tính chẵn lẻ, và việc chia đôi có thể được thực hiện trong thời gian đơn vị.

33-2 Phân tích các phép toán bit trong thuật toán Euclid

a. Chứng tỏ việc dùng thuật toán "giấy và bút chì" bình thường với phép chia dài—chia a với b, cho ra một số thương q và số dư r—yêu cầu $O((1 + \lg q) \lg b)$ phép toán bit.

b. Định nghĩa $\mu(a, b) = (1 + \lg a)(1 + \lg b)$. Chứng tỏ số lượng các phép toán bit được thực hiện bởi EUCLID để rút gọn bài toán tính toán $\gcd(a, b)$ thành bài toán tính toán $\gcd(b, a \bmod b)$ tối đa là $c(\mu(a, b) - \mu(b, a \bmod b))$ với một hằng đủ lớn $c > 0$.

c. Chứng tỏ EUCLID(a, b) yêu cầu $O(\mu(a, b))$ phép toán bit nói chung và $O(\beta^2)$ phép toán bit khi được áp dụng cho hai đầu vào β-bit.

33-3 Ba thuật toán cho các số Fibonacci

Bài toán này so sánh hiệu năng của ba phương pháp để tính toán số Fibonacci thứ n F_n, đã cho n. Giả sử mức hao phí của việc cộng, trừ, hoặc nhân hai con số là $O(1)$, bậcc lập với kích cỡ của các con số.

a. Chứng tỏ thời gian thực hiện của phương pháp đệ quy đơn giản để tính toán F_n dựa trên phép truy toán (2.13) là hàm số mũ trong n.

b. Nêu cách tính toán F_n trong $O(\lg n)$ thời gian dùng phép memo hóa [memoization].

c. Nêu cách tính toán F_n trong $O(\lg n)$ thời gian chỉ dùng phép cộng và phép nhân số nguyên. (*Mách nước:* Xét ma trận

$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

và các lũy thừa của nó.)

d. Giờ đây mặc nhận tiến trình cộng hai số β-bit mất $\Theta(\beta)$ thời gian và tiến trình nhân hai số β-bit mất $\Theta(\beta^2)$ thời gian. Nêu thời gian thực hiện của ba phương pháp này dưới số đo mức hao phí hợp lý hơn này cho các phép toán số học cơ bản?

33-4 Các thặng dư bình phương

Cho p là một số nguyên tố lẻ. Một số $a \in Z^*_p$ là một ***thặng dư bình phương*** nếu phương trình $x^2 = a \pmod p$ có một giải pháp cho ẩn số x.

a. Chứng tỏ có chính xác $(p-1)/2$ thặng dư bình phương, modulo p.

b. Nếu p là số nguyên tố, ta định nghĩa ***ký hiệu Legendre*** $(\frac{a}{p})$, với $a \in Z^*_p$, là 1 nếu a là một thặng dư bình phương modulo p và bằng không là -1. Chứng minh nếu $a \in Z^*_p$, thì

$$\left(\frac{a}{p}\right) \equiv a^{(p-1)/2} \pmod p\ .$$

Nêu một thuật toán hiệu quả để xác định một số a đã cho là một thặng dư bình phương modulo p. Phân tích hiệu năng của thuật toán.

c. Chứng minh nếu p là một số nguyên tố có dạng $4k+3$ và a là một thặng dư bình phương trong Z^*_p, thì $a^{k+1} \bmod p$ là một căn bậc hai của a, modulo p. Nêu lượng thời gian cần thiết để tìm căn bậc hai của một thặng dư bình phương a modulo p?

d. Mô tả một thuật toán hiệu quả ngẫu nhiên hóa để tìm một thặng dư không bình phương, modulo một số nguyên tố tùy ý p. Thuật toán của bạn yêu cầu trung bình bao nhiêu phép toán số học?

Ghi chú Chương

Niven và Zuckerman [151] cung cấp phần giới thiệu tuyệt vời về lý thuyết số cơ bản. Knuth [122] chứa một phần thảo luận hay về các thuật toán để tìm ước số chung lớn nhất, cũng như các thuật toán lý thuyết số cơ bản khác. Riesel [168] và Bach [16] cung cấp các cuộc khảo sát mới hơn về lý thuyết số tính toán. Dixon [56] nêu khái quát phép thừa số hóa và phép thử tính nguyên. Tập báo cáo hội nghị được Pomerance [159] hiệu chỉnh có chứa vài bài nghiên cứu hay.

Knuth [122] mô tả nguồn gốc của thuật toán Euclid. Nó xuất hiện trong Tập 7, Định đề 1 và 2, của cuốn *Các thành phần* của nhà toán học Hy lạp Euclid, đã được viết khoảng năm 300 B.C. Phần mô tả của Euclid có thể đã được phái sinh từ một thuật toán của Eudoxus khoảng năm 375 B.C. Thuật toán của Euclid có thể tự hào là thuật toán không hiển nhiên xưa nhất; nó chỉ được sánh với thuật toán nông dân Nga về phép

nhân (xem Chương 29), được xem là của người Ai-cập cổ.

Knuth quy một trường hợp đặc biệt của định lý phần dư Trung Quốc (Định lý 33.27) cho nhà toán học Trung hoa Sun-Tsu, sống vào khoảng giữa năm 200 B.C. và 200 A.D. –ngày tháng không thật chắc chắn. Cùng trường hợp đặc biệt đó đã được đưa ra bởi nhà toán học Hy Lạp Nichomachus vào khoảng năm 100 A.D. Nó đã được tổng quát hóa bởi Chin Chiu-Shao vào năm 1247. Định lý số dư Trung Quốc cuối cùng đã được L. Euler phát biểu và chứng minh với đầy đủ tính tổng quát của nó vào năm 1734.

Thuật toán thử tính nguyên ngẫu nhiên hóa được trình bày ở đây là của Miller [147] và Rabin [166]; nó là thuật toán thử tính nguyên ngẫu nhiên hóa nhanh nhất được biết, trong các thừa số bất biến. Phần chứng minh của Định lý 33.39 là một thích ứng sơ bộ của phần chứng minh do Bach [15] đề xuất. Monier [148, 149] cung cấp một chứng minh về một kết quả mạnh hơn của MILLER-RABIN. Để có được thuật toán thử tính nguyên thời gian đa thức, dường như ta cần phải dùng phép ngẫu nhiên hóa. Thuật toán thử tính nguyên tất định được biết là phiên bản Cohen-Lenstra [45] của phép thử tính nguyên của Adleman, Pomerance, và Rumely [3]. Khi thử tính nguyên một số n có chiều dài $\lceil \lg(n+1) \rceil$, nó chạy trong $(\lg n)^{O(\lg\lg\lg n)}$ thời gian, đúng là hơi siêu đa thức.

Bài toán tìm các số nguyên tố "ngẫu nhiên" lớn được mô tả chi tiết trong một bài viết của Beauchemin, Brassard, Crepeau, Goutier, và Pomerance [20].

Khái niệm về một hệ mật mã khóa công là do Diffie và Hellman [54]. Hệ mật mã RSA đã được đề xuất bởi Rivest, Shamir, và Adleman [169] vào năm 1977. Từ đó trở đi, lĩnh vực mật mã hóa đã nở hoa. Nói cụ thể, nhiều kỹ thuật mới đã được phát triển để chứng minh các hệ mật mã là an ninh. Ví dụ, Goldwasser và Micali [86] chứng tỏ phép ngẫu nhiên hóa có thể là một công cụ hiệu quả để thiết kế các lược đồ mật mã hóa khóa công an ninh. Với các lược đồ chữ ký, Goldwasser, Micali, và Rivest [88] trình bày một lược đồ chữ ký số hóa mà mọi kiểu giả mạo có thể tưởng tượng được cũng khó như việc lấy thừa số. Gần đây, Goldwasser, Micali, và Rackoff [87] đã giới thiệu một lớp lược đồ mật mã hóa "kiến thức zero" qua đó người ta có thể chứng minh (dưới một số giả thiết hợp lý nhất định) rằng không bên nào biết được nhiều hơn mức dự trù được biết từ một cuộc truyền thông.

Heuristic rho để lấy thừa số số nguyên đã được Pollard [156] sáng

chế. Phiên bản được trình bày ở đây là một biến thức do Brent [35] đề nghị.

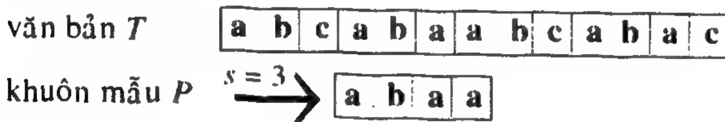
Các thuật toán tốt nhất để lấy thừa số các con số lớn có một thời gian thực hiện đại khái tăng trưởng theo hàm mũ với căn bậc hai của chiều dài của số n được lấy thừa số. Thuật toán lấy thừa số sàng bình phương [quadratic-sieve factoring] của Pomerance [158], nói chung có lẽ là thuật toán hiệu quả nhất như vậy dành cho các đầu vào lớn. Mặc dù khó lòng đưa ra một phân tích nghiêm ngặt về thuật toán này, song dưới giả thiết hợp lý ta có thể suy ra một ước lượng thời gian thực hiện của $L(n)^{1+o(1)}$, ở đó $L(n) = e^{\sqrt{\ln n \ln \ln n}}$. Với vài đầu vào, phương pháp đường cong êlip của Lenstra [137] có thể hiệu quả hơn so với phương pháp sàng bình phương, bởi, giống như phương pháp rho của Pollard, nó có thể tìm ra một thừa số nguyên tố nhỏ p khá nhanh chóng. Với phương pháp này, thời gian để tìm p được ước lượng là $L(p)^{\sqrt{2}+o(1)}$.

34 So Khớp Chuỗi

Việc tìm tất cả các lần xuất hiện của một khuôn mẫu trong một văn bản là một bài toán thường nảy sinh trong các chương trình hiệu chỉnh văn bản. Thông thường, văn bản là một tư liệu đang được hiệu chỉnh, và khuôn mẫu tìm kiếm là một từ cụ thể do người dùng cung cấp. Các thuật toán hiệu quả cho bài toán này có thể hỗ trợ đắc lực cho sự phản ứng nhanh nhạy của chương trình hiệu chỉnh văn bản. Ví dụ, các thuật toán so khớp chuỗi cũng được dùng để tìm kiếm các khuôn mẫu cụ thể trong các dãy DNA.

Ta hình thức hóa ***bài toán so khớp chuỗi*** như sau. Mặc nhận rằng văn bản là một mảng $T[1..n]$ có chiều dài n và khuôn mẫu là một mảng $P[1..m]$ có chiều dài m. Cũng mặc nhận thêm rằng các thành phần của P và T là các ký tự được rút từ một bảng chữ cái hữu hạn Σ. Ví dụ, ta có thể có $\Sigma = \{0, 1\}$ hoặc $\Sigma = \{a, b, \ldots, z\}$. Các mảng ký tự P và T thường được gọi là các ***chuỗi*** ký tự.

Ta nói rằng khuôn mẫu ***P xảy ra với khóa chuyển*** s trong văn bản T (hoặc, theo tương đương, nói rằng khuôn mẫu ***P xảy ra bắt đầu tại vị trí s + 1*** trong văn bản T) nếu $0 \leq s \leq n - m$ và $T[s + 1..s + m] = P[1..m]$ (nghĩa là, nếu $T[s + j] = P[j]$, với $1 \leq j \leq m$). Nếu P xảy ra với khóa chuyển s trong T, thì ta gọi s là một ***khóa chuyển hợp lệ***; bằng không, ta gọi s là một ***khóa chuyển không hợp lệ***. Bài toán so khớp chuỗi là bài toán tìm tất cả các khóa chuyển hợp lệ với nó một khuôn mẫu P đã cho xảy ra trong một văn bản T đã cho. Hình 34.1 minh họa các phần định nghĩa này.



Hình 34.1 Bài toán so khớp chuỗi. Mục tiêu đó là tìm ra tất cả các lần xuất hiện của khuôn mẫu P = abaa trong văn bản T = abcabaabcabac. Khuôn mẫu xảy ra chỉ một lần trong văn bản, tại khóa chuyển $s = 3$. Khóa chuyển $s = 3$ được gọi là một khóa chuyển hợp lệ. Mỗi ký tự của khuôn mẫu được nối bằng một vạch dọc với ký tự so khớp trong văn bản, và tất cả các ký tự so khớp được nêu ở dạng tô bóng.

Chương này được tổ chức như sau. Trong Đoạn 34.1 ta ôn lại thuật toán cưỡng bức đơn sơ cho bài toán so khớp chuỗi, có thời gian thực hiện trường hợp xấu nhất $O((n - m + 1)m)$. Đoạn 34.2 trình bày một thuật toán so khớp chuỗi đáng quan tâm của Rabin và Karp. Thuật toán này cũng có thời gian thực hiện trường hợp xấu nhất $O((n - m + 1)m)$, nhưng tính trung bình và trong thực tế nó làm việc tốt hơn nhiều. Nó cũng tổng quát hóa tốt theo các bài toán so khớp khuôn mẫu khác. Đoạn 34.3 mô tả một thuật toán so khớp chuỗi bắt đầu bằng cách kiến tạo một otomat hữu hạn được thiết kế cụ thể để tìm kiếm các lần xuất hiện của khuôn mẫu P đã cho trong một văn bản. Thuật toán này chạy trong thời gian $O(n + m|\Sigma|)$. Thuật toán Knuth-Morris-Pratt (hoặc KMP) tương tự nhưng thông minh hơn nhiều được trình bày trong Đoạn 34.4; thuật toán KMP chạy trong thời gian $O(n + m)$. Cuối cùng, Đoạn 34.5 mô tả một thuật toán của Boyer và Moore mà thường là chọn lựa thực tiễn nhất, mặc dù thời gian thực hiện trường hợp xấu nhất của nó (giống như của thuật toán Rabin-Karp) không tốt hơn thuật toán so khớp chuỗi đơn sơ.

Hệ ký hiệu và thuật ngữ

Ta sẽ cho Σ^* (đọc là "sigma-star") thể hiện tập hợp tất cả các chuỗi có chiều dài hữu hạn được hình thành nhờ dùng các ký tự từ bảng chữ cái Σ. Trong chương này, ta chỉ xét các chuỗi có chiều dài hữu hạn. Chuỗi trống có chiều dài zero, được ký hiệu là e, cũng thuộc về Σ^*. Chiều dài của một chuỗi x được ký hiệu $|x|$. ***Phép ghép nối*** hai chuỗi x và y, được ký hiệu là xy, có chiều dài $|x| + |y|$ và bao gồm các ký tự từ x theo sau là các ký tự từ y.

Ta nói rằng một chuỗi w là ***tiền tố*** của một chuỗi x, được ký hiệu là $w \sqsubset x$, nếu $x = wy$ với một chuỗi $y \in \Sigma^*$. Lưu ý, nếu $w \sqsubset x$, thì $|w| \le |x|$. Cũng vậy, ta nói rằng một chuỗi w là một ***hậu tố*** của một chuỗi x, được ký hiệu là $w \sqsupset x$, nếu $x = yw$ với một $y \in \Sigma^*$. Do $w \sqsupset x$ mà có $|w| \le |x|$. Chuỗi trống a là hậu tố lẫn tiền tố của mọi chuỗi. Ví dụ, ta có ab $\sqsubset$ abcca và cca $\sqsupset$ abcca. Cũng nên lưu ý với bất kỳ các chuỗi x và y và bất kỳ ký tự a, ta có $x \sqsupset y$ nếu và chỉ nếu $xa \sqsupset ya$. Ngoài ra, cũng lưu ý $\sqsubset$ và $\sqsupset$ là các hệ thức bắc cầu. Bổ đề dưới đây sẽ hữu ích về sau.

Bổ đề 34.1 (Bổ đề phủ chồng hậu tố)

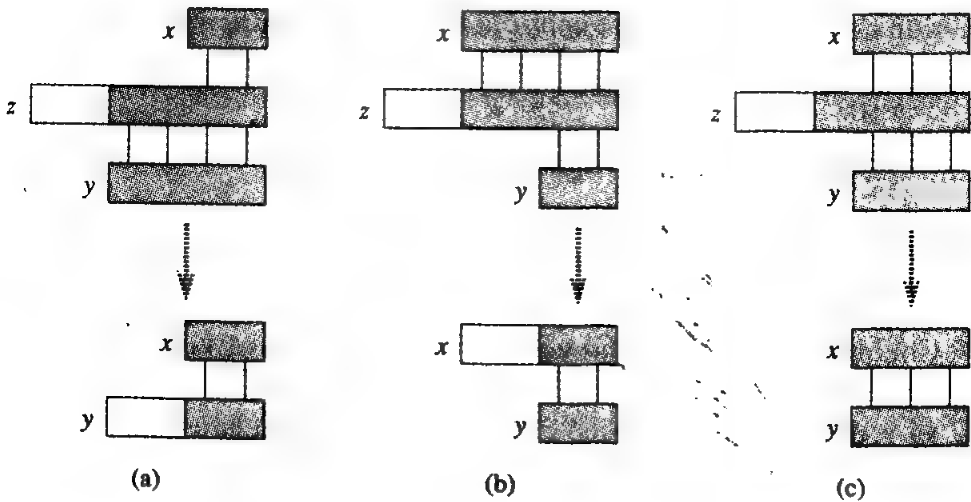
Giả sử x, y, và z là các chuỗi sao cho $x \sqsupset z$ và $y \sqsupset z$. Nếu $|x| \le |y|$, thì $x \sqsupset y$. Nếu $|x| \ge |y|$, thì $y \sqsupset x$. Nếu $|x| = |y|$, thì $x = y$.

Chứng minh Xem Hình 34.2 để có phần chứng minh đồ họa.

Để ngắn gọn về hệ ký hiệu, ta sẽ dùng P_k để thể hiện tiền tố k-ký tự

$P[1..k]$ của khuôn mẫu $P[1..m]$. Như vậy, $P_0 = \varepsilon$ và $P_m = P = P[1..m]$. Cũng vậy, ta dùng T_k để thể hiện tiền tố k-ký tự của văn bản T. Dùng hệ ký hiệu này, ta có thể phát biểu bài toán so khớp chuỗi dưới dạng bài toán tìm tất cả các khóa chuyển s trong miền giá trị $0 \le s \le n - m$ sao cho $P \sqsupset T_{s+m}$.

Trong mã giả của chúng ta, ta cho phép hai chuỗi có chiều dài bằng nhau được so sánh về đẳng thức dưới dạng một phép toán nguyên thủy. Nếu các chuỗi được so sánh từ trái qua phải và phép so sánh dừng khi gặp một trường hợp không so khớp, ta mặc nhận rằng thời gian dành cho một đợt trắc nghiệm như vậy là một hàm tuyến tính của số lượng các ký tự so khớp đã được khám phá. Để chính xác, đợt kiểm tra "$x = y$" được mặc nhận chiếm thời gian $\Theta(t + 1)$, ở đó t là chiều dài của chuỗi dài nhất z sao cho $z \sqsubset x$ và $z \sqsubset y$.



Hình 34.2 Một chứng minh đồ họa của Bổ đề 34.1. Ta giả sử rằng $x \sqsupset z$ và $y \sqsupset z$, Ba phần của hình minh họa ba trường hợp của bổ đề. Các vạch dọc nối các vùng so khớp (được tô bóng) của các chuỗi. **(a)** Nếu $|x| \le |y|$, thì $x \sqsupset y$. **(b)** Nếu $|x| \ge |y|$, thì $y \sqsupset x$. **(c)** Nếu $|x| = |y|$, thì $x = y$.

34.1 Thuật toán so khớp chuỗi đơn sơ

Thuật toán đơn sơ tìm tất cả các khóa chuyển hợp lệ dùng một vòng lặp kiểm tra điều kiện $P[1..m] = T[s + 1..s + m]$ với mỗi trong số $n - m + 1$ giá trị khả dĩ của s.

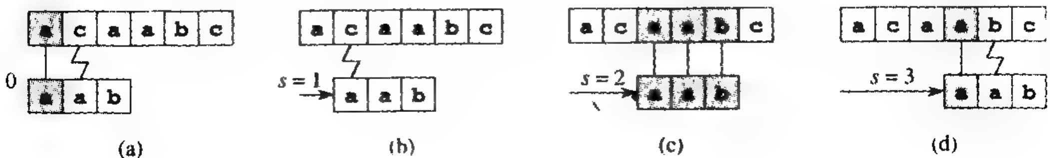
NAIVE-STRING-MATCHER(T, P)

1 $n \leftarrow length[T]$

2 $m \leftarrow length[P]$

```
3 for s ← 0 to n – m
4     do if P[1..m] = T[s + 1..s + m]
5         then in "Khuôn mẫu xảy ra với khóa chuyển" s
```

Theo đồ họa, thủ tục so khớp chuỗi đơn sơ có thể được diễn dịch dưới dạng một "tập mẫu" chứa khuôn mẫu trượt trên văn bản, ghi chú về các khóa chuyển mà tất cả các ký tự trên tập mẫu bằng với các ký tự tương ứng trong văn bản, như minh họa trong Hình 34.3. Vòng lặp **for** bắt đầu trên dòng 3 xét từng khóa chuyển khả dĩ một cách rõ rệt. Đợt kiểm tra trên dòng 4 xác định khóa chuyển hiện hành có hợp lệ hay không; đợt kiểm tra này có liên quan đến một vòng lặp mặc định để kiểm tra các vị trí ký tự tương ứng cho đến khi tất cả các vị trí so khớp thành công hoặc tìm thấy một trường hợp không so khớp. Dòng 5 in ra từng khóa chuyển hợp lệ s.



Hình 34.3 Phép toán của bộ so khớp chuỗi đơn sơ với khuôn mẫu P = aab và văn bản T = acaabc. Ta có thể tưởng tượng khuôn mẫu P dưới dạng một "tập mẫu" mà ta trượt cạnh văn bản. Các phần (a)-(d) nêu bốn kiểu căn thẳng hàng liên tiếp mà bộ so khớp chuỗi đơn sơ thử. Trong mỗi phần, các vạch dọc nối các vùng tương ứng tìm thấy so khớp (được tô bóng), và một vạch zích zắc nối ký tự không so khớp đầu tiên tìm thấy, nếu có. Một trường hợp xuất hiện của khuôn mẫu được tìm thấy, tại khóa chuyển s = 2, nêu trong phần (c).

Thủ tục NAIVE-STRING-MATCHER mất một thời gian $\Theta((n - m + 1)m)$ trong trường hợp xấu nhất. Ví dụ, xét văn bản chuỗi a^n (một chuỗi gồm n a) và khuôn mẫu a^m. Với mỗi trong số $n - m + 1$ giá trị khả dĩ của khóa chuyển s, vòng lặp mặc định trên dòng 4 để so sánh các ký tự tương ứng phải thi hành m lần để phê chuẩn khóa chuyển. Như vậy, thời gian thực hiện trường hợp xấu nhất là $\Theta((n - m + 1)m)$, sẽ là $\Theta(n^2)$ nếu $m = \lfloor n/2 \rfloor$.

Như sẽ thấy, NAIVE-STRING-MATCHER không phải là một thủ tục tối ưu cho bài toán này. Quả vậy, trong chương này ta sẽ nêu một thuật toán có thời gian thực hiện trường hợp xấu nhất là $O(n+m)$. Bộ so khớp chuỗi đơn sơ không hiệu quả bởi vì thông tin thu được về văn bản với một giá trị của s sẽ hoàn toàn được bỏ qua trong khi xét các giá trị khác của s. Tuy nhiên, thông tin như vậy có thể rất quý giá. Ví dụ, nếu P = aaab và ta thấy rằng s = 0 là hợp lệ, thì không có khóa chuyển nào

trong số 1, 2, hoặc 3 là hợp lệ, bởi $T[4]$ = b. Trong các đoạn dưới đây, ta xét vài cách để vận dụng hiệu quả kiểu sắp xếp thông tin này.

Bài tập

34.1-1

Nêu các phép so sánh mà bộ so khớp chuỗi đơn sơ thực hiện cho khuôn mẫu P = 0001 trong văn bản T = 000010001010001.

34.1-2

Chứng tỏ thời gian trường hợp xấu nhất để bộ so khớp chuỗi đơn sơ tìm ra trường hợp xuất hiện đầu tiên của một khuôn mẫu trong một văn bản là $\Theta((n - m + 1)(m - 1))$.

34.1-3

Giả sử tất cả các ký tự trong khuôn mẫu P đều khác nhau. Nêu cách gia tốc NAIVE-STRING-MATCHER để chạy trong thời gian $O(n)$ trên một văn bản n-ký tự T.

34.1-4

Giả sử khuôn mẫu P và văn bản T là các chuỗi được chọn *ngẫu nhiên* có chiều dài m và n, theo thứ tự nêu trên, từ bảng chữ cái thuộc d $\Sigma_d = \{0, 1, ..., d - 1\}$, ở đó $d \geq 2$. Chứng tỏ số lượng dự trù các phép so sánh theo từng ký tự mà vòng lặp mặc định thực hiện trong dòng 4 của thuật toán đơn sơ là

$$(n - m + 1)\frac{1 - d^{-m}}{1 - d^{-1}} \leq 2(n - m + 1).$$

(Giả sử thuật toán đơn sơ ngừng so sánh các ký tự với một khóa chuyển đã cho một khi tìm thấy một trường hợp không so khớp hoặc nguyên cả khuôn mẫu đều so khớp.) Như vậy, với các chuỗi được chọn ngẫu nhiên, thuật toán đơn sơ khá hiệu quả.

34.1-5

Giả sử ta cho phép khuôn mẫu P chứa các lần xuất hiện của một ***ký tự khe hở*** ◊ có thể so khớp một chuỗi các ký tự tùy ý (thậm chí một chuỗi có chiều dài zero).

Ví dụ, khuôn mẫu ab ◊ ba ◊ c xảy ra trong văn bản cabccbacbacab dưới dạng

c ab cc ba cba c ab

ab ◊ ba ◊ c

và dưới dạng

$c\,\underbrace{ab}_{ab}\,\underbrace{ccbac}_{\Diamond}\,\underbrace{ba}_{ba}\,\underbrace{\quad}_{\Diamond}\,\underbrace{c}_{c}\,ab\ .$

Lưu ý, ký tự khe hở có thể xảy ra theo một số lần tùy ý trong khuôn mẫu nhưng được mặc nhận không xảy ra trong văn bản. Nêu thuật toán thời gian đa thức để xác định xem một khuôn mẫu P như vậy có xảy ra trong một văn bản đã cho T, hay không, và phân tích thời gian thực hiện của thuật toán.

34.2 Thuật toán Rabin-Karp

Rabin và Karp đã đề xuất một thuật toán so khớp chuỗi thực hiện tốt trong thực tế và cũng tổng quát hóa theo các thuật toán khác cho các bài toán có liên quan, như khuôn mẫu so khớp hai chiều. Thời gian thực hiện trường hợp xấu nhất của thuật toán Rabin-Karp là $O((n - m + 1)m)$, nhưng nó có một thời gian thực hiện trường hợp trung bình tốt.

Thuật toán này vận dụng các khái niệm lý thuyết số cơ bản như sự tương đương của hai số modulo một số thứ ba. Nếu cần, bạn tham khảo Đoạn 33.1 để có các phần định nghĩa hữu quan.

Để dễ giải thích, ta mặc nhận $\Sigma = \{0, 1, 2, ..., 9\}$, sao cho mỗi ký tự là một chữ số thập phân. (Trong trường hợp chung, ta có thể mặc nhận mỗi ký tự là một chữ số theo hệ ký hiệu cơ số-d, ở đó $d = |\Sigma|$.) Như vậy, ta có thể xem một chuỗi gồm k ký tự liên tiếp dưới dạng biểu thị một số thập phân có chiều dài k. Như vậy, chuỗi ký tự 31415 tương ứng với số thập phân 31,415. Căn cứ vào phép diễn dịch đối ngẫu các ký tự đầu vào dưới dạng các ký hiệu đồ họa lẫn các chữ số, để tiện dụng, trong đoạn này ta thể hiện chúng như ta vẫn làm với các chữ số, theo phông chữ văn bản chuẩn của chúng ta.

Cho một khuôn mẫu $P[1..m]$, ta cho p thể hiện giá trị thập phân tương ứng của nó. Một cách tương tự, cho một văn bản $T[1..n]$, ta cho t_s thể hiện giá trị thập phân của chuỗi con có chiều dài-m $T[s+1..s+m]$, với $s = 0, 1, ..., n - m$. Tất nhiên, $t_s = p$ nếu và chỉ nếu $T[s + 1..s + m] = P[1..m]$; như vậy, s là một khóa chuyển hợp lệ nếu và chỉ nếu $t_s = p$. Nếu ta có thể tính toán p trong thời gian $O(m)$ và tất cả các giá trị t_s trong một tổng $O(n)$ thời gian, thì ta có thể xác định tất cả các khóa chuyển hợp lệ s trong thời gian $O(n)$ bằng cách so sánh p với mỗi trong số các t_s. (Trước mắt, đừng quan tâm về khả năng mà p và các t_s có thể là các con số rất lớn.)

Ta có thể tính toán p trong thời gian $O(m)$ dùng quy tắc Horner (xem

Đoạn 32.1): $p = P[m] + 10\ (P[m-1] + 10(P[m-2] + ... + 10(P[2] + 10P[1])\ ...))$. Giá trị t_0 có thể được tính toán một cách tương tự từ $T[1..m]$ trong thời gian $O(m)$.

Để tính toán các giá trị còn lại $t_1, t_2, ..., t_{n-m}$ trong thời gian $O(n - m)$, ta chỉ cần nhận thấy t_{s+1}, có thể được tính toán từ t_s trong thời gian bất biến, bởi

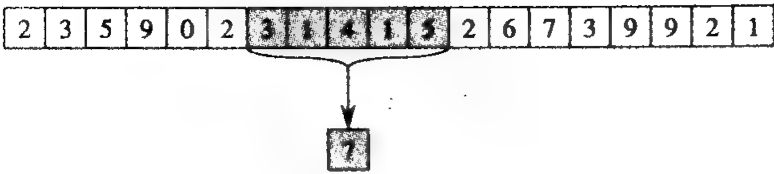
$$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1]\ . \qquad (34.1)$$

Ví dụ, nếu $m = 5$ và $t_s = 31415$, thì ta muốn gỡ bỏ chữ số thứ tự cao $T[s+1] = 3$ và mang vào chữ số thứ tự thấp mới (giả sử nó là $T[s+5+1] = 2$) để có

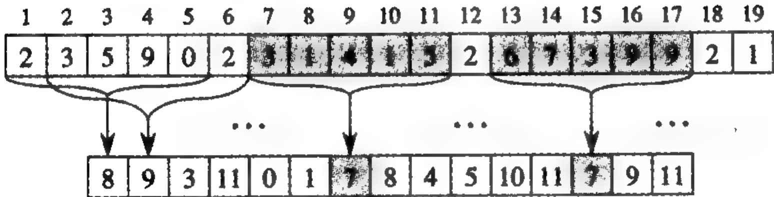
$$\begin{aligned} t_{s+1} &= 10(31415 - 10000 \bullet 3) + 2 \\ &= 14152\ . \end{aligned}$$

Trừ $10^{m-1}T[s+1]$ để gỡ bỏ chữ số thứ tự cao ra khỏi t_s, nhân kết quả với 10 để chuyển con số sang trái một vị trí, và cộng $T[s+m+1]$ để mang vào chữ số thứ tự thấp thích hợp. Nếu hằng 10^{m-1} được tính toán trước (có thể thực hiện trong thời gian $O(\lg m)$ dùng các kỹ thuật của Đoạn 33.6, mặc dù với ứng dụng này một phương pháp $O(m)$ đơn giản tỏ ra khá thích đáng), thì mỗi lần thi hành của phương trình (34.1) sẽ chiếm một số lượng bất biến các phép toán số học. Như vậy, tất cả p và $t_0, t_1, ..., t_{n-m}$ đều có thể được tính toán trong thời gian $O(n+m)$, và ta có thể tìm ra tất cả các lần xuất hiện của khuôn mẫu $P[1..m]$ trong văn bản $T[1..n]$ trong thời gian $O(n+m)$.

Khó khăn duy nhất với thủ tục này đó là p và t_s có thể quá lớn để tiện làm việc. Nếu P chứa m ký tự, thì việc mặc nhận mỗi phép toán số học trên p (dài m chữ số) chiếm "thời gian bất biến" là không hợp lý. May thay, có một cách chữa đơn giản cho bài toán này, như đã nêu trong Hình 34.4, đó là: tính toán p và các t_s modulo một modulus q thích hợp. Bởi phép tính của p, t_0, và phép truy toán (34.1) tất cả đều có thể được thực hiện modulo q, ta thấy rằng p và tất cả các t_s có thể được tính toán modulo q trong thời gian $O(n+m)$. Modulus q thường được chọn dưới dạng một số nguyên tố sao cho $10q$ chỉ vừa trong một từ máy tính, cho phép tất cả các phép tính cần thiết được thực hiện bằng số học chính xác đơn [single- precision].



(a)



so khớp hợp lệ

lần đụng giả mạo

chữ số thứ tự cao cũ

chữ số thứ tự thấp mới

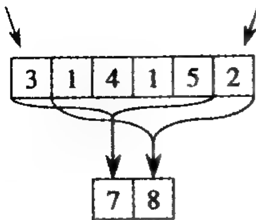
(b)

chữ số thứ tự cao cũ

khóa chuyển

chữ số thứ tự thấp mới

old mới old mới



$$\begin{aligned} 14152 &\equiv (31415 - 3 \cdot 10000) \cdot 10 + 2 \pmod{13} \\ &\equiv (7 - 3 \cdot 3) \cdot 10 + 2 \pmod{13} \\ &\equiv 8 \pmod{13} \end{aligned}$$

(c)

Hình 34.4 Thuật toán Rabin Karp. Mỗi ký tự là một chữ số thập phân, và ta tính toán các giá trị modulo 13. **(a)** Một chuỗi văn bản. Một cửa sổ có chiều dài 5 được nêu ở dạng tô bóng. Giá trị số của con số tô bóng được tính toán modulo 13, cho ra giá trị 7. **(b)** Cùng chuỗi văn bản có các giá trị đã tính toán modulo 13 với mỗi vị trí khả dĩ của một cửa sổ có chiều dài 5. Giả sử khuôn mẫu $P = 31415$, ta tìm các cửa sổ có giá trị modulo 13 là 7, bởi $31415 \equiv 7 \pmod{13}$. Hai cửa sổ như vậy được tìm thấy, được nêu ở dạng tô bóng trong hình. Cửa sổ đầu tiên, bắt đầu tại vị trí văn bản 7, quả thực là một trường hợp xuất hiện của khuôn mẫu, trong khi cửa sổ thứ hai, bắt đầu tại vị trí văn bản 13, là một lần đụng giả mạo. **(c)** Tính toán giá trị của một cửa sổ trong thời gian bất biến, căn cứ vào giá trị của cửa sổ trước đó. Cửa sổ đầu tiên có giá trị 31415. Thải bỏ chữ số thứ tự cao 3, chuyển sang trái (nhân cho 10), rồi cộng vào chữ số thứ tự thấp 2 để cho ra giá trị mới 14152. Tuy nhiên, tất cả các phép tính được thực hiện modulo 13, do đó giá trị của cửa sổ đầu tiên là 7, và giá trị được tính toán cho cửa sổ mới là 8.

Nói chung, với một bảng chữ cái thuộc d {0,1,..., $d-1$}, ta chọn q sao cho dq vừa trong một từ máy tính và điều chỉnh phương trình truy toán (34.1) để làm việc modulo q, sao cho nó trở thành

$$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q\,, \qquad (34.2)$$

ở đó $h \equiv d^{m-1} \pmod q$ là giá trị của chữ số "1" trong vị trí thứ tự cao của một cửa sổ văn bản m-chữ số.

Tuy nhiên, nồi canh làm việc modulo q giờ đây có chứa một chú sâu, bởi $t_s \equiv p \pmod q$ không hàm ý rằng $t_s = p$. Mặt khác, nếu $t_s \not\equiv p$ (mod q), thì ta dứt khoát có $t_s \neq p$, sao cho khóa chuyển s là không hợp lệ. Như vậy, ta có thể dùng đợt kiểm tra $t_s \equiv p \pmod q$ dưới dạng một đợt kiểm tra heuristic nhanh để loại trừ các khóa chuyển không hợp lệ s. Mọi khóa chuyển s mà $t_s \equiv p \pmod q$ đều phải được kiểm tra thêm để xem s có thực sự hợp lệ hay không hay ta chỉ có một ***lần đụng giả mạo***. Để thực hiện đợt kiểm tra này, ta có thể tiến hành kiểm tra cụ thể điều kiện $P[1..m] = T[s+1..s+m]$. Nếu q đủ lớn, ta có thể hy vọng các lần đụng giả mạo xảy ra không thường xuyên đủ để mức hao phí của việc kiểm tra phụ trội ở mức thấp.

Thủ tục dưới đây biến các ý tưởng này trở nên chính xác. Các đầu vào cho thủ tục là văn bản T, khuôn mẫu P, cơ số d để sử dụng (thường được lấy là $|\Sigma|$), và số nguyên tố q để sử dụng.

```
RABIN-KARP-MATCHER(T, P, d, q)
n ← length[T]
m ← length[P]
h ← d^(m-1) mod q
p ← 0
t_0 ← 0
for i ← 1 to m
      do p ← (dp + P[i]) mod q
         t_0 ← (dt_0 + T[i]) mod q
for s ← 0 to n - m
      do if p = t_s
            then if P[1..m] = T[s + 1..s + m]
                    then "Pattern xảy ra with khóa chuyển" s
         if s < n - m
            then t_(s+1) ← (d(t_s - T[s + 1]h) + T[s + m + 1]) mod q
```

Thủ tục RABIN-KARP-MATCHER làm việc như sau. Tất cả các ký tự được diễn dịch dưới dạng các chữ số cơ số-d. Các con chữ dưới trên t được cung cấp chỉ để minh bạch; chương trình làm việc đúng đắn nếu tất cả các con chữ dưới được thải bỏ. Dòng 3 khởi tạo h theo giá trị của vị trí chữ số thứ tự cao của một cửa sổ m-chữ số. Các dòng 4-8 tính toán p dưới dạng giá trị của $P[1..m]$ mod q và t_0 dưới dạng giá trị của $T[1..m]$ mod q. Vòng lặp **for** bắt đầu trên dòng 9 lặp lại qua tất cả các khóa chuyển s khả dĩ. Vòng lặp có sự bất biến sau đây: mỗi khi dòng 10 được thi hành, $t_s = T[s + 1..s + m]$ mod q. Nếu $p = t_s$ trong dòng 10 (một "lần đụng"), thì ta kiểm tra để xem có $P[1..m] = T[s + 1..s + m]$ trong dòng 11 hay không để loại trừ khả năng của một lần đụng giả mạo. Mọi khóa chuyển hợp lệ tìm thấy đều được in ra trên dòng 12. Nếu $s < n - m$ (được kiểm tra trong dòng 13), thì vòng lặp **for** sẽ được thi hành ít nhất một lần nữa, và do đó dòng 14 được thi hành trước để bảo đảm vẫn duy trì sự bất biến vòng lặp khi đụng lại dòng 10. Dòng 14 tính toán giá trị của t_{s+1} mod g từ giá trị của t_s mod q trong thời gian bất biến dùng trực tiếp phương trình (34.2).

Thời gian thực hiện của RABIN-KARP-MATCHER là $\Theta((n - m + 1)m)$ trong trường hợp xấu nhất, bởi (giống như thuật toán so khớp chuỗi đơn sơ) thuật toán Rabin-Karp xác minh rõ rệt mọi khóa chuyển hợp lệ. Nếu $P = a^m$ và $T = a^n$, thì các lần xác minh chiếm thời gian $\Theta((n - m + 1)m)$, bởi mỗi trong số $n - m + 1$ khóa chuyển khả dĩ là hợp lệ. (Cũng lưu ý rằng phép tính của d^{m-1} mod q trên dòng 3 và vòng lặp trên các dòng 6-8 chiếm thời gian $O(m) = O((n - m + 1)m)$.)

Trong nhiều ứng dụng, ta dự kiến ít có khóa chuyển hợp lệ (có lẽ $O(1)$ trong số chúng), và do đó thời gian thực hiện dự trù của thuật toán là $O(n + m)$ cộng với thời gian cần thiết để xử lý các lần đụng giả mạo. Ta có thể đặt sự phân tích heuristic dựa trên cơ sở giả thiết rút gọn các giá trị modulo q tác động như một phép ánh xạ ngẫu nhiên từ Σ^* đến $\mathbf{Z}_q$. (Xem phần mô tả về cách dùng phép chia cho kỹ thuật ánh số trong Đoạn 12.3.1. Ta khó lòng hình thức hóa và chứng minh một giả thiết như vậy, mặc dù một cách tiếp cận có thể tồn tại đó là mặc nhận rằng q được chọn một cách ngẫu nhiên từ các số nguyên có kích cỡ thích hợp. Ở đây, ta sẽ không theo đuổi kiểu hình thức hóa này.) Như vậy, ta có thể dự kiến số lượng các lần đụng giả mạo là $O(n/q)$ bởi cơ may mà một t_s tùy ý sẽ tương đương với p, modulo q, có thể được ước lượng là $1/q$. Như vậy, thời lượng dự trù của thuật toán Rabin Karp là

$$O(n) + O(m(v + n/q)),$$

ở đó v là số lượng các khóa chuyển hợp lệ. Thời gian thực hiện này là $O(n)$ nếu ta chọn $q \geq m$. Nghĩa là, nếu số lượng dự trù của các khóa chuyển hợp lệ là nhỏ ($O(1)$) và số nguyên tố q được chọn lớn hơn chiều dài của khuôn mẫu, thì ta có thể dự kiến thủ tục Rabin-Karp sẽ chạy trong thời gian $O(n + m)$.

Bài tập

34.2-1

Làm việc modulo $q = 11$, bộ so khớp Rabin-Karp gặp bao nhiêu các lần đụng giả mạo trong văn bản $T = 3141592653589793$ khi tìm khuôn mẫu $P = 26$?

34.2-2

Nếu cách mở rộng phương pháp Rabin-Karp cho bài toán lục trong một chuỗi văn bản để tìm một lần xuất hiện của bất kỳ một trong số một tập hợp k khuôn mẫu đã cho?

34.2-3

Nêu cách mở rộng phương pháp Rabin-Karp để điều quản bài toán tìm một khuôn mẫu $m \times m$ đã cho trong một mảng $n \times n$ các ký tự. (Khuôn mẫu có thể được chuyển dọc và ngang, nhưng nó không thể quay.)

34.2-4

Alice có một bản sao về một tập tin dài n-bit $A = \langle a_{n-1}, a_{n-2}, \ldots, a_0 \rangle$, và Bob cũng có một tập tin n-bit $B = \langle b_{n-1}, b_{n-2}, \ldots, b_0 \rangle$. Alice và Bob muốn biết xem các tập tin của họ có đồng nhất hay không. Để tránh truyền toàn bộ A hoặc B, họ dùng đợt kiểm tra theo xác suất nhanh dưới đây. Cùng nhau, họ lựa một số nguyên tố $q > 1000n$ và ngẫu nhiên lựa một số nguyên x từ $\{0, 1, \ldots, n - 1\}$. Sau đó, Alice đánh giá

$$A(x) = \left(\sum_{i=0}^{n} a_i x^i \right) \bmod q$$

và Bob cũng đánh giá $B(x)$. Chứng minh nếu $A \neq B$, sẽ có tối đa một cơ may trong 1000 rằng $A(x) = B(x)$, trong khi đó nếu hai tập tin giống nhau, $A(x)$ nhất thiết giống như $B(x)$. (*Mách nước:* Xem Bài tập 33.4-4.)

34.3 So khớp chuỗi với otomat hữu hạn

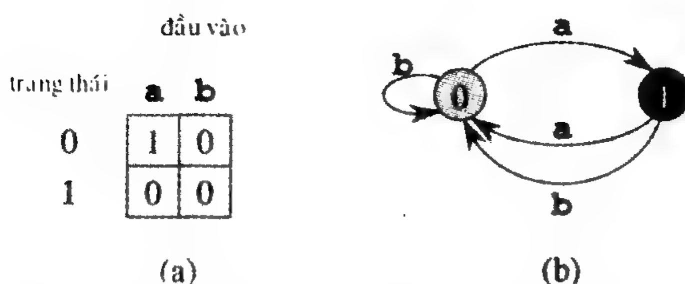
Nhiều thuật toán so khớp chuỗi xây dựng một otomat hữu hạn quét văn bản chuỗi T với tất cả các lần xuất hiện của khuôn mẫu P. Đoạn này trình bày một phương pháp để xây dựng một otomat như vậy. Otomat so khớp chuỗi này rất hiệu quả: chúng xét mỗi ký tự văn bản *chính xác một lần,* trải qua thời gian bất biến với mỗi ký tự văn bản. Do đó, thời gian được dùng—sau khi xây dựng otomat—là $\Theta(n)$. Tuy nhiên, thời gian để xây dựng otomat có thể lớn nếu Σ lớn. Đoạn 34.4 mô tả một cách thông minh để giải quyết bài toán này.

Ta bắt đầu đoạn này bằng phần định nghĩa của một otomat hữu hạn. Sau đó, ta xét một otomat so khớp chuỗi đặc biệt và nêu cách dùng nó để tìm các lần xuất hiện của một khuôn mẫu trong một văn bản. Phần mô tả này bao gồm các chi tiết về cách mô phỏng cách ứng xử của một otomat so khớp chuỗi trên một văn bản đã cho. Cuối cùng, ta sẽ nêu cách kiến tạo otomat so khớp chuỗi cho một khuôn mẫu đầu vào đã cho.

Otomat hữu hạn

Một otomat hữu hạn M là một bộ-5 [5-tuple] $(Q, q_0, A, \Sigma, \delta)$, ở đó

- Q là một tập hợp hữu hạn ***các trạng thái***,



Hình 34.5 Một otomat hữu hạn hai trạng thái đơn giản với tập hợp trạng thái $Q = \{0, 1\}$, trạng thái đầu $q_0 = 0$, và bảng chữ cái đầu vào $\Sigma = \{a, b\}$. **(a)** Một phần biểu diễn bảng của hàm chuyển tiếp δ. **(b)** Một sơ đồ chuyển tiếp trạng thái tương đương. Trạng thái 1 là trạng thái chấp nhận duy nhất (được tô đen). Các cạnh có hướng biểu diễn các bước chuyển tiếp. Ví dụ, cạnh từ trạng thái 1 đến trạng thái 0 được gán nhãn b nêu rõ $\delta(1, b) = 0$. Otomat này chấp nhận các chuỗi kết thúc trong một số lẻ các a. Chính xác hơn, một chuỗi x được chấp nhận nếu và chỉ nếu $x = yz$, ở đó $y = \varepsilon$ hoặc y kết thúc bằng a b, và $z = a^k$, ở đó k là lẻ. Ví dụ, dãy các trạng thái mà otomat này nhập cho đầu vào abaaa (kể cả trạng thái đầu) là <0, 1, 0 1, 0, 1>, và do đó nó chấp nhận đầu vào này. Với đầu vào abbaa, dãy các trạng thái là <0, 1, 0, 0, 1, 0>, và do đó nó loại bỏ đầu vào này.

- $q_0 \in Q$ là ***trạng thái đầu***,
- $A \subseteq Q$ là một tập hợp đặc biệt của ***các trạng thái chấp nhận***.
- Σ là một ***bảng chữ cái đầu vào*** hữu hạn.
- δ là một hàm từ $Q \times \Sigma$ vào Q, có tên ***hàm chuyển tiếp*** của M.

Otomat hữu hạn bắt đầu trong trạng thái q_0 và lần lượt đọc từng ký tự chuỗi đầu vào của nó. Nếu otomat nằm trong trạng thái q và đọc ký tự đầu vào a, nó dời ("thực hiện một bước chuyển tiếp") từ trạng thái q sang trạng thái $\delta(q, a)$. Mỗi khi trạng thái hiện hành q của nó là một phần tử của A, máy M được xem là đã ***chấp nhận*** chuỗi đã đọc cho đến giờ. Một đầu vào không được chấp nhận được xem là bị ***loại bỏ***. Hình 34.5 minh họa các phần định nghĩa này bằng một otomat hai-trạng thái đơn giản.

Một otomat hữu hạn M cảm sinh một hàm ϕ, có tên ***hàm trạng thái chung cuộc***, từ Σ^* đến Q sao cho $\phi(w)$ là trạng thái mà M kết thúc sau khi quét t chuỗi w. Như vậy, M chấp nhận một chuỗi w nếu và chỉ nếu $\phi(w) \in A$. Hàm ϕ được định nghĩa bởi hệ thức đệ quy

$$\phi(\varepsilon) = q_0 ,$$

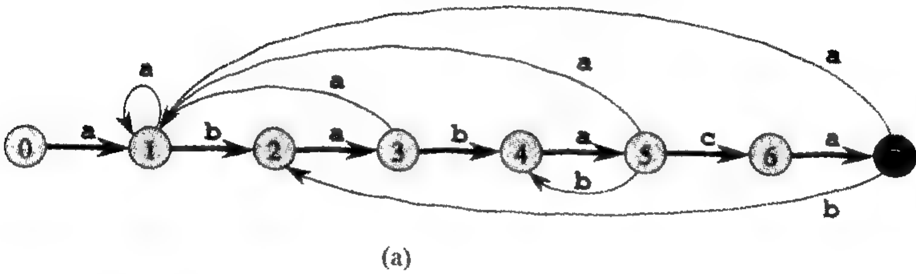
$$\phi(wa) = \delta(\phi(w), a) \qquad \text{for } w \in \Sigma^*,\ a \in \Sigma .$$

Otomat so khớp chuỗi

Có một otomat so khớp chuỗi cho mọi khuôn mẫu P; otomat này phải được kiến tạo từ khuôn mẫu trong một bước tiền xử lý trước khi dùng nó để tìm trong chuỗi văn bản. Hình 34.6 minh họa cách kiến tạo này với khuôn mẫu P = ababaca. Từ đây trở đi, ta mặc nhận P là một chuỗi khuôn mẫu cố định đã cho; để ngắn gọn, ta sẽ không nêu rõ sự phụ thuộc vào P trong hệ ký hiệu của chúng ta.

Để chỉ định otomat so khớp chuỗi tương ứng với một khuôn mẫu đã cho $P[1..m]$, trước tiên ta định nghĩa một hàm phụ σ, có tên ***hàm hậu tố*** tương ứng với P. Hàm σ là một phép ánh xạ từ Σ^* đến $\{0, 1, ..., m\}$ sao cho $\sigma(x)$ là chiều dài của tiền tố dài nhất của P là một hậu tố của x:

$$\sigma(x) = \max\{k : P_k \sqsupset x\} .$$



(a)

trạng thái	đầu vào a	b	c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

(b)

i	—	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	—	a	b	a	b	a	b	a	c	a	b	a
state $\phi(T_i)$	0	1	2	3	4	5	4	5	6	7	2	3

(c)

Hình 34.6 (a) Một sơ đồ chuyển tiếp trạng thái của otomat so khớp chuỗi chấp nhận tất cả các chuỗi kết thúc trong chuỗi ababaca. Trạng thái 0 là trạng thái đầu, và trạng thái 7 (được tô đen) là trạng thái chấp nhận duy nhất. Một cạnh có hướng từ trạng thái i đến trạng thái j được gắn nhãn a biểu diễn $\delta(i, a) = j$. Các cạnh sang phải hình thành "cột sống" của otomat, được nêu đậm trong hình, tương ứng với các lần so khớp thành công giữa khuôn mẫu và các ký tự đầu vào. Các cạnh sang phải tương ứng các lần so khớp thất bại. Có vài cạnh tương ứng với các lần so khớp thất bại không được nêu; theo quy ước, nếu một trạng thái i không có cạnh đi ra được gắn nhãn a với một $a \in \Sigma$, thì $\delta(i, a) = 0$. **(b)** Hàm chuyển tiếp tương ứng δ, và chuỗi khuôn mẫu P = ababaca. Các khoản nhập tương ứng với các lần so khớp thành công giữa khuôn mẫu và các ký tự đầu vào được nêu ở dạng tô bóng. **(c)** Phép toán của otomat trên văn bản T = abababacaba. Dưới mỗi ký tự văn bản $T[i]$ được gán trạng thái $\phi(T_i)$ mà otomat nằm trong đó sau khi xử lý tiền tố T_i. Một lần xuất hiện của khuôn mẫu được tìm thấy, kết thúc tại vị trí 9.

Hàm hậu tố σ được định nghĩa kỹ bởi chuỗi trống $P_0 = \varepsilon$ là một hậu tố của mọi chuỗi. Ví dụ, với khuôn mẫu P = ab, ta có $\sigma(\varepsilon) = 0$, σ(ccaca) = 1, và σ(ccab) = 2. Với một khuôn mẫu P có chiều dài m, ta có $\sigma(x) = m$ nếu và chỉ nếu $P \sqsupset x$. Từ phần định nghĩa của hàm hậu tố, ta có nếu $x \sqsupset y$, thì $\sigma(x) \leq \sigma(y)$.

Ta định nghĩa otomat so khớp chuỗi tương ứng với một khuôn mẫu đã cho $P[1..m]$ như sau.

- Tập hợp trạng thái Q là $\{0, 1, ..., m\}$. Trạng thái đầu q_0 là trạng thái 0, và trạng thái m là trạng thái chấp nhận duy nhất.
- Hàm chuyển tiếp δ được định nghĩa bởi phương trình dưới đây, với bất kỳ trạng thái q và ký tự a:

$$\delta(q, a) = \sigma(P_q a) . \tag{34.3}$$

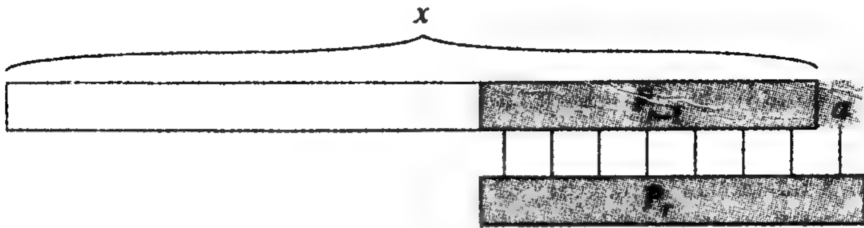
Đây là một nhân tố căn bản trực giác cho định nghĩa $\delta(q, a) = \sigma(P_q a)$. Máy duy trì dưới dạng một bất biến về hoạt động củanó rằng

$$\phi(T_i) = \sigma(T_i) ; \tag{34.4}$$

kết quả này được chứng minh như Định lý 34.4 dưới đây. Tóm lại, điều này có nghĩa là sau khi quét i ký tự đầu tiên của chuỗi văn bản T, máy nằm trong trạng thái $\phi(T_i) = q$, ở đó $q = \sigma(T_i)$ là chiều dài của hậu tố dài nhất của T_i cũng là một tiền tố của khuôn mẫu P. Nếu ký tự kế tiếp được quét là $T(i + 1) = a$, thì máy sẽ thực hiện một bước chuyển tiếp sang trạng thái $\sigma(T_{i+1}) = \sigma(T_i a)$. Phần chứng minh của định lý chứng tỏ $\sigma(T_i a) = \sigma(P_q a)$. Nghĩa là, để tính toán chiều dài của hậu tố dài nhất của $T_i a$ là một tiền tố của P, ta có thể tính toán hậu tố dài nhất của $P_q a$ là một tiền tố của P. Tại mỗi trạng thái, máy chỉ cần biết chiều dài của tiền tố dài nhất của P là một hậu tố của nội dung đã được đọc cho đến giờ. Do đó, việc ấn định $\delta(q, a) = \sigma(P_q a)$ duy trì sự bất biến mong muốn (34.4). Dưới đây ta sẽ thấy lập luận không chính thức này là chính xác.

Ví dụ, trong otomat so khớp chuỗi của Hình 34.6, ta có $\delta(5, b) = 4$. Điều này là do sự việc nếu otomat đọc a b trong trạng thái $q = 5$, thì P_qb = ababab, và tiền tố dài nhất của P that cũng là một hậu tố của ababab là P_4 = abab.

Để làm rõ phép toán của một otomat so khớp chuỗi, giờ đây ta cho một vị trí đơn giản, hiệu quả để mô phỏng cách ứng xử của một otomat như vậy (được biểu diễn bằng hàm chuyển tiếp δ của nó) trong khi tìm các lần xuất hiện của một khuôn mẫu P có chiều dài m trong một văn bản đầu vào $T[1..n]$. Cũng như với bất kỳ otomat so khớp chuỗi nào dành cho một khuôn mẫu có chiều dài m, tập hợp trạng thái Q là $\{0, 1, ..., m\}$, trạng thái đầu là 0, và trạng thái chấp nhận duy nhất là trạng thái m.



Hình 34.7 Một minh họa cho phần chứng minh của Bổ đề 34.2. Hình chứng tỏ $r \leq \sigma(x) + 1$, ở đó $r = \sigma(xa)$.

```
FINITE-AUTOMATON-MATCHER(T, δ, m)
n ← length[T]
q ← 0
for i ← 1 to n
        do q ← δ(q, T[i])
            if q = m
                then s ← i - m
                    print "Khuôn mẫu xảy ra với khóa chuyển" s
```

Cấu trúc vòng lặp đơn giản của FINITE-AUTOMATON-MATCHER hàm ý rằng thời gian thực hiện của nó trên một chuỗi văn bản có chiều dài n là $O(n)$. Tuy nhiên, thời gian thực hiện này không gộp thời gian cần có để tính toán hàm chuyển tiếp δ. Ta sẽ đề cập bài toán này sau, sau khi chứng minh thủ tục FINITE-AUTOMATON-MATCHER hoạt động đúng đắn.

Xét phép toán của otomat trên một văn bản đầu vào $T[1..n]$. Ta sẽ chứng minh otomat nằm trong trạng thái $\sigma(T_i)$ sau khi quét ký tự $T[i]$. Bởi $\sigma(T_i) = m$ nếu và chỉ nếu $P \sqsupset T_i$, máy nằm trong trạng thái chấp nhận m nếu và chỉ nếu khuôn mẫu P vừa được quét. Để chứng minh kết quả này, ta vận dụng hai bổ đề dưới đây về σ.

Bổ đề 34.2 (Bất đẳng thức hàm hậu tố)

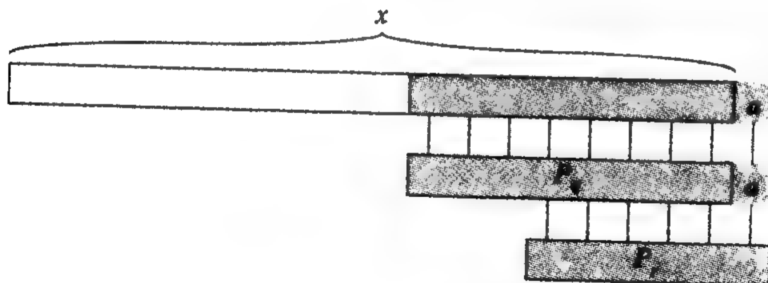
Với bất kỳ chuỗi x và ký tự a, ta có $\sigma(xa) \leq \sigma(x) + 1$.

Chứng minh Tham khảo Hình 34.7, cho $r = \sigma(xa)$. Nếu $r = 0$, thì kết luận $r \leq \sigma(x) + 1$ là thỏa đáng một cách tầm thường, theo tính không phủ định của $\sigma(x)$. Do đó mặc nhận rằng $r > 0$. Giờ đây, $P_r \sqsupset xa$, theo định nghĩa của σ. Như vậy, $P_{r-1} \sqsupset x$, bằng cách bỏ a ra khỏi cuối P_r và ra khỏi cuối xa. Do đó, $r - 1 \leq \sigma(x)$, bởi $\sigma(x)$ là k lớn nhất sao cho $P_k \sqsupset x$.

Bổ đề 34.3 (Bổ đề đệ quy hàm hậu tố)

Với bất kỳ chuỗi x và ký tự a, nếu $q = \sigma(x)$, thì $\sigma(xa) = \sigma(P_q a)$.

Chứng minh Từ phần định nghĩa của σ, ta có $P_q \sqsupset x$. Như Hình 34.8 đã nêu, $P_q a \sqsupset xa$. Nếu ta cho $r = \sigma(xa)$, thì $r \leq q + 1$ theo Bổ đề 34.2. Bởi $P_q a \sqsupset xa$, $P_r \sqsupset xa$, và $|P_r| \leq |P_q a|$, Bổ đề 34.1 hàm ý rằng $P_r \sqsupset P_q a$. Do đó, $r \leq \sigma(P_q a)$, nghĩa là, $\sigma(xa) \leq \sigma(P_q a)$. Nhưng ta cũng có $\sigma(P_q a) \leq \sigma(xa)$, bởi $P_q a \sqsupset xa$. Như vậy, $\sigma(xa) = \sigma(P_q a)$.



Hình 34.8 Một minh họa cho phần chứng minh của Bổ đề 34.3. Hình chứng tỏ $r = \sigma(P_q a)$, ở đó $q = \sigma(x)$ và $r = \sigma(xa)$.

Giờ đây ta đã sẵn sàng chứng minh định lý chính nêu đặc trưng cách ứng xử của một otomat so khớp chuỗi trên một văn bản đầu vào đã cho. Như đã nêu trên đây, định lý này chứng tỏ tại mỗi bước, otomat đơn thuần theo dõi tiền tố dài nhất của khuôn mẫu là một hậu tố của nội dung đã được đọc cho đến giờ.

Định lý 34.4

Nếu ϕ là hàm trạng thái chung cuộc của một otomat so khớp chuỗi với một khuôn mẫu đã cho P và $T[1..n]$ là một văn bản đầu vào cho otomat, thì

$$\phi(T_i) = \sigma(T_i)$$

với $i = 0, 1, \ldots, n$.

Chứng minh Phần chứng minh là theo phương pháp quy nạp trên i. Với $i = 0$, định lý là đúng một cách tầm thường, bởi $T_0 = \varepsilon$. Như vậy, $\phi(T_0) = \sigma(T_0) = 0$.

Giờ đây, ta mặc nhận rằng $\phi(T_i) = \sigma(T_i)$ và chứng minh rằng $\phi(T_{i+1}) = \sigma(T_{i+1})$. Cho q thể hiện $\phi(T_i)$, và cho a thể hiện $T[i + 1]$. Thì,

$$\begin{aligned}
\phi(T_{i+1}) &= \phi(T_i a) && \text{(theo các phần định nghĩa của } T_{i+1} \text{ và } a) \\
&= \delta(\phi(T_i), a) && \text{(theo phần định nghĩa của } \phi) \\
&= \delta(q, a) && \text{(theo phần định nghĩa của } q)
\end{aligned}$$

$= \sigma(P_q a)$ (theo phần định nghĩa (34.3) của δ)

$= \sigma(T_i a)$ (theo Bổ đề 34.3 và phương pháp quy nạp)

$= \sigma(T_{i+1})$ (theo phần định nghĩa của T_{i+1}) .

Theo phương pháp quy nạp, định lý được chứng minh.

Theo Định lý 34.4, nếu máy nhập trạng thái q trên dòng 4, thì q là giá trị lớn nhất sao cho $P_q \sqsupset T_i$. Như vậy, ta có $q = m$ trên dòng 5 nếu và chỉ nếu một trường hợp xuất hiện của khuôn mẫu P vừa được quét. Ta kết luận rằng FINITE-AUTOMATON-MATCHER hoạt động đúng đắn.

Tính toán hàm chuyển tiếp

Thủ tục dưới đây tính toán hàm chuyển tiếp δ từ một khuôn mẫu đã cho $P[1..m]$.

COMPUTE-TRANSITION-FUNCTION(P, Σ)

```
1   m ← length[P]
2   for q ← 0 to m
3•      do for mỗi ký tự a ∈ Σ
4              do k ← min(m + 1, q + 2)
5                 repeat k ← k − 1
6                        until P_k ⊐ P_q a
7                 δ(q, a) ← k
8   return δ
```

Thủ tục này tính toán $\delta(q, a)$ một cách đơn giản theo định nghĩa của nó. Các vòng lặp lồng nhau bắt đầu trên các dòng 2 và 3 xét tất cả các trạng thái q và các ký tự a, và các dòng 4-7 ấn định $\delta(q, a)$ là k lớn nhất sao cho $P_k \sqsupset P_q a$. Mã bắt đầu với giá trị có thể mường tượng được lớn nhất của k, là $\min(m, q + 1)$, và giảm k cho đến khi $P_k \sqsupset P_q a$.

Thời gian thực hiện của COMPUTE-TRANSITION-FUNCTION là $O(m^3|\Sigma|)$, bởi các vòng lặp phía ngoài đóng góp một thừa số của m $|\Sigma|$, vòng lặp **repeat** phía trong có thể chạy tối đa $m + 1$ lần, và đợt kiểm tra $P_k \sqsupset P_q a$ trên dòng 6 có thể yêu cầu so sánh tới m ký tự. Hiện có các thủ tục nhanh hơn nhiều; thời gian cần thiết để tính toán δ từ P có thể được cải thiện thành $O(m|\Sigma|)$ bằng cách vận dụng vài thông tin đã tính toán một cách thông minh về khuôn mẫu P (xem Bài tập 34.4-6). Với thủ tục đã cải tiến này để tính toán δ, tổng thời gian thực hiện để tìm tất cả các lần xuất hiện của một khuôn mẫu có chiều dài-m trong một văn bản có chiều dài-n trên một bảng chữ cái Σ là $O(n + m|\Sigma|)$.

Bài tập

34.3-1

Kiến tạo otomat so khớp chuỗi cho khuôn mẫu P = aabab và minh họa phép toán củanó trên chuỗi văn bản T = aaababaabaababaab.

34.3-2

Vẽ một sơ đồ chuyển tiếp trạng thái cho một otomat so khớp chuỗi với khuôn mẫu ababbabbababbababbabb trên bảng chữ cái $\Sigma = \{a, b\}$.

34.3-3

Ta gọi một khuôn mẫu P là ***không thể phủ chồng*** [nonoverlappable] nếu $P_k \sqsupset P_q$ hàm ý $k = 0$ hoặc $k = q$. Mô tả sơ đồ chuyển tiếp trạng thái của otomat so khớp chuỗi với một khuôn mẫu không thể phủ chồng.

34.3-4 *

Cho hai khuôn mẫu P và P', mô tả cách kiến tạo một otomat hữu hạn xác định tất cả các lần xuất hiện của *một trong hai* khuôn mẫu. Gắng giảm thiểu số lượng trạng thái trong otomat của bạn.

34.3-5

Cho một khuôn mẫu P chứa các ký tự khe hở (xem Bài tập 34.1-5), nêu cách xây dựng một otomat hữu hạn có thể tìm ra một lần xuất hiện của P trong một văn bản T trong $O(n)$ thời gian, ở đó $n = |T|$.

34.4 Thuật toán Knuth-Morris-Pratt

Giờ đây, ta trình bày một thuật toán so khớp chuỗi thời gian tuyến tính của Knuth, Morris, và Pratt. Thuật toán của họ đạt một thời gian thực hiện $\Theta(n + m)$ bằng cách tránh phép tính của hàm chuyển tiếp δ, và nó thực hiện khuôn mẫu so khớp bằng chỉ một hàm phụ $\pi[1..m]$ được tính toán trước từ khuôn mẫu trong thời gian $O(m)$. Mảng π cho phép tính toán hàm chuyển tiếp δ một cách hiệu quả (theo nghĩa khấu trừ) "ngay lập tức" theo yêu cầu. Nói nôm na, với một trạng thái $q = 0, 1, \ldots, m$ và một ký tự $a \in \Sigma$, giá trị $\pi[q]$ chứa thông tin độc lập của a và cần có để tính toán $\delta(q, a)$. (Ghi chú này sẽ được làm rõ dưới đây.) Bởi mảng π chỉ có m khoản nhập, trong khi đó δ có $O(m|\Sigma|)$ khoản nhập, ta lưu một thừa số của Σ trong tiến trình xử lý trước bằng cách tính toán π thay vì δ.

Hàm tiền tố cho một khuôn mẫu

Hàm tiền tố của một khuôn mẫu gói riêng kiến thức về cách thức mà khuôn mẫu so khớp đối lại với các khóa chuyển của chính nó. Có thể dùng thông tin này để tránh việc trắc nghiệm các khóa chuyển vô dụng trong thuật toán so khớp khuôn mẫu đơn sơ hoặc tránh phép tính trước của δ với một otomat so khớp chuỗi.

Xét phép toán của bộ so khớp chuỗi đơn sơ. Hình 34.9(a) nêu một khóa chuyển cụ thể s của một tập mẫu chứa khuôn mẫu P = ababaca đối lại một văn bản T. Với ví dụ này, $q = 5$ ký tự đã so khớp thành công, nhưng ký tự khuôn mẫu thứ 6 thất bại trong việc so khớp ký tự văn bản tương ứng. Thông tin mà q ký tự đã so khớp thành công sẽ xác định các ký tự văn bản tương ứng. Biết được q ký tự văn bản này sẽ cho phép ta xác định ngay một số khóa chuyển là không hợp lệ. Trong ví dụ của hình, các khóa chuyển $s + 1$ nhất thiết là không hợp lệ, bởi ký tự khuôn mẫu đầu tiên, một a, sẽ được căn thẳng hàng với một ký tự văn bản được biết là so khớp với ký tự khuôn mẫu thứ hai, a b. Tuy nhiên, khóa chuyển $s + 2$ nêu trong phần (b) của hình căn thẳng hàng ba ký tự khuôn mẫu đầu tiên với ba ký tự văn bản phải nhất thiết so khớp. Nói chung, ta nên biết câu trả lời cho câu hỏi sau:

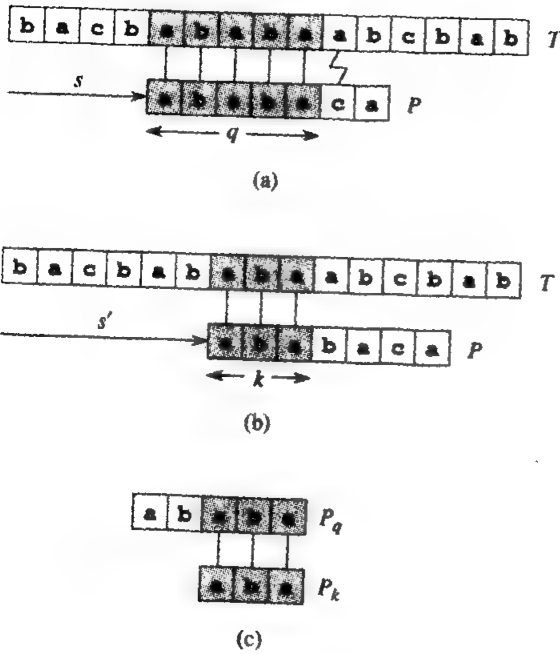
Cho các ký tự khuôn mẫu $P[1..q]$ so khớp các ký tự văn bản $T[s + 1 .. s + q]$, đâu là khóa chuyển bé nhất $s' > s$ sao cho

$$P[1..k] = T[s' + 1..s' + k], \tag{34.5}$$

ở đó $s' + k = s + q$?

Một khóa chuyển s' như vậy chính là khóa chuyển đầu tiên lớn hơn s không nhất thiết là không hợp lệ do kiến thức của chúng ta về $T[s + 1..s + q]$. Trong trường hợp tốt nhất, ta có $s' = s + q$, và tất cả các khóa chuyển $s + 1, s + 2, ..., s + q - 1$ lập tức được loại trừ. Trong mọi trường hợp, tại khóa chuyển mới s' ta không cần so sánh k ký tự đầu tiên của P với các ký tự tương ứng của T, bởi ta được bảo đảm chúng so khớp theo phương trình (34.5).

Thông tin cần thiết có thể được tính toán trước bằng cách so sánh khuôn mẫu với chính nó, như minh họa trong Hình 34.9(c). Bởi $T[s'+ 1..s'+ k]$ là thành phần của phần văn bản đã biết, nên nó là một hậu tố của chuỗi P_q. Do đó, phương trình (34.5) có thể được diễn dịch như đang yêu cầu $k < q$ lớn nhất sao cho $P_k \sqsupset P_q$. Như vậy, $s' = s + (q - k)$ là khóa chuyển có tiềm năng hợp lệ kế tiếp. Sự thể hóa ra tiện dụng khi



Hình 34.9 Hàm tiền tố π. **(a)** Khuôn mẫu P = ababaca được căn thẳng hàng với một văn bản T sao cho $q = 5$ ký tự đầu tiên so khớp. Việc so khớp các ký tự, được tô bóng, được nối bởi các vạch dọc. **(b)** Chỉ dùng kiến thức của chúng ta về 5 ký tự đã so khớp, ta có thể suy ra rằng một khóa chuyển của $s+1$ không hợp lệ, nhưng một khóa chuyển của $s' = s + 2$ nhất quán với mọi thứ mà ta know về văn bản và do đó có tiềm năng hợp lệ. **(c)** Thông tin hữu ích cho các phép suy diễn như vậy có thể được tính toán trước bằng cách so sánh khuôn mẫu với chính nó. Ở đây, ta thấy rằng tiền tố dài nhất của P cũng là một hậu tố của P_5 là P_3. Thông tin này được tính toán trước và được biểu diễn trong mảng π, sao cho $\pi[5] = 3$. Cho q ký tự đã so khớp thành công tại khóa chuyển s, khóa chuyển có tiềm năng hợp lệ kế tiếp nằm tại $s' = s + (q - \pi[q])$.

lưu trữ số k ký tự so khớp tại khóa chuyển s' mới, thay vì lưu trữ, giả sử, $s' - s$. Thông tin này có thể được dùng để tăng tốc cả thuật toán so khớp chuỗi đơn sơ lẫn bộ so khớp otomat hữu hạn.

Ta hình thức hóa việc tính toán trước cần thiết như sau. Cho một khuôn mẫu $P[1..m]$, hàm tiền tố cho khuôn mẫu P là hàm $\pi : \{1,2,...,m\} \rightarrow \{0, 1,...,m - 1\}$ sao cho

$$\pi[q] = \max\{k : k < q \text{ và } P_k \sqsupset P_q\}.$$

Nghĩa là, $\pi[q]$ là chiều dài của tiền tố dài nhất của P là một hậu tố riêng của P_q. Để lấy một ví dụ khác, Hình 34.10(a) cung cấp hàm tiền tố hoàn thành π cho khuôn mẫu ababababca.

Thuật toán so khớp Knuth-Morns-Pratt được cung cấp trong mã giả dưới đây dưới dạng thủ tục KMP-MATCHER. Như sẽ thấy, nó chủ yếu

được mô hình hóa theo FINITE-AUTOMATON-MATCHER. KMP-MATCHER gọi thủ tục phụ COMPUTE-PREFIX-FUNCTION để tính toán π.

```
KMP-MATCHER(T, P)
n ← length[T]
m ← length[P]
π ← COMPUTE-PREFIX-FUNCTION(P)
q ← 0
for i ← to n
      do while q > 0 và P[q + 1] ≠ T[i]
                  do q ← π[q]
         if P[q + 1] = T[i]
            then q ← q + 1
         if q = m
            then print "Pattern occurs with shift" i – m
                  q ← π[q]
```

```
COMPUTE-PREFIX-FUNCTION(P)
m ← length[P]
π[1] ← 0
k ← 0
for q ← 2 to m
      do while k > 0 và P[k + 1 ] ≠ P[q]
               do k ← π[k]
         if P[k + 1] = P[q]
            then k ← k + 1
         π[q] ← k
return π
```

Ta bắt đầu bằng một phân tích về thời gian thực hiện của các thủ tục này. Việc chứng minh các thủ tục này đúng đắn sẽ phức tạp hơn.

Phân tích thời gian thực hiện

Thời gian thực hiện của COMPUTE-PREFIX-FUNCTION là $O(m)$, dùng một phân tích khấu trừ (xem Chương 18). Ta kết nợp một thế [potential] của k với trạng thái hiện hành k của thuật toán. Thế này có

một giá trị ban đầu là 0, theo dòng 3. Dòng 6 giảm k mỗi khi nó được thi hành, bởi $\pi[k] < k$. Tuy nhiên, do $\pi[k] \geq 0$ với tất cả k, nên k có thể không bao giờ trở thành âm. Dòng khác duy nhất tác động đến k là dòng 8, gia tăng k tối đa là một trong mỗi lần thi hành của thân vòng lặp **for**. Bởi $k < q$ trong khi nhập vòng lặp **for**, và bởi q được gia số trong mỗi lần lặp lại của thân vòng lặp **for**, nên $k < q$ luôn đứng vững. (Điều này chứng minh cho biện luận rằng $\pi[q] < q$, theo dòng 9.) Ta có thể thanh toán cho mỗi lần thi hành của thân vòng lặp **while** trên dòng 6 bằng một mức giảm tương ứng trong hàm thế, bởi $\pi[k] < k$. Dòng 8 gia tăng hàm thế tối đa là một, sao cho mức hao phí khấu trừ của thân vòng lặp trên các dòng 5-9 là $O(1)$. Bởi số lần lặp lại của vòng lặp phía ngoài là $O(m)$, và bởi hàm thế chung cuộc ít nhất cũng lớn bằng hàm thế ban đầu, nên tổng thời gian thực hiện thực tế trường hợp xấu nhất của COMPUTE-PREFIX-FUNCTION là $O(m)$.

Thuật toán Knuth-Morris-Pratt chạy trong thời gian $O(m + n)$. Lệnh gọi của COMPUTE-PREFIX-FUNCTION chiếm $O(m)$ thời gian như ta vừa thấy, và một phân tích khấu trừ tương tự, dùng giá trị của q làm hàm thế, chứng tỏ phần còn lại của KMP-MATCHER chiếm $O(n)$ thời gian.

So sánh với FINITE-AUTOMATON-MATCHER, nhờ dùng π thay vì δ, ta đã rút gọn thời gian để tiền xử lý khuôn mẫu từ $O(m|\Sigma|)$ đến $O(m)$, trong khi vẫn giữ thời gian so khớp thực tế được định cận bởi $O(m + n)$.

Tính đúng đắn của phép tính hàm tiền tố

Ta bắt đầu bằng một bổ đề thiết yếu chứng tỏ rằng bằng cách lặp lại hàm tiền tố π, ta có thể điểm danh tất cả các tiền tố P_k là các hậu tố của một tiền tố P_q đã cho. Cho

$\pi^*[q] = \{q, \pi[q], \pi^2[q], \pi^3[q], ..., \pi^t[q]\}$

ở đó $\pi^i[q]$ được định nghĩa theo dạng sự cấu thành chức năng, sao cho $\pi^0[q] = q$ và $\pi^{i+1}[q] = \pi[\pi^i[q]]$ với $i > 1$, và ở đó nó được hiểu là dãy trong $\pi^*[q]$ sẽ dừng khi đụng $\pi^t[q] = 0$.

Bổ đề 34.5 (Bổ đề lặp lại hàm tiền tố)

Cho P là một khuôn mẫu có chiều dài m với hàm tiền tố π. Thì, với $q = 1, 2, ..., m$, ta có $\pi^*[q] = \{k : P_k \sqsupset P_q\}$.

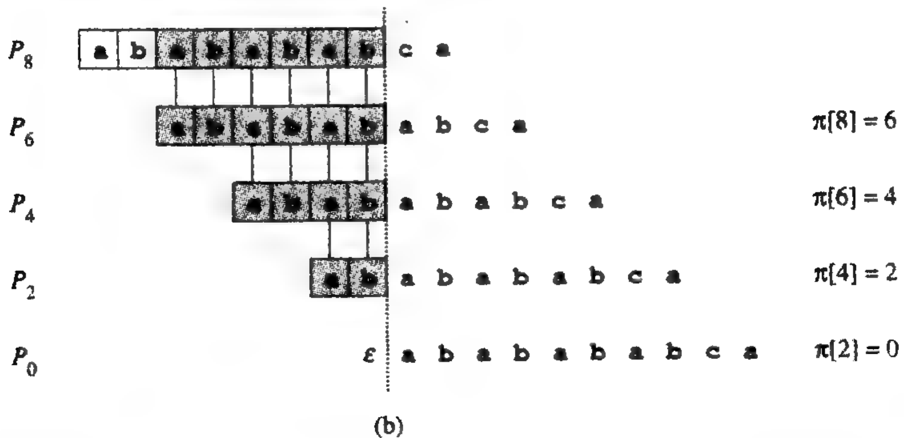
Chứng minh Trước tiên ta chứng minh rằng

$i \in \pi^*[q]$ hàm ý $P_i \sqsupset P_q$. (34.6)

Nếu $i \in \pi^*[q]$, thì $i = \pi^u[q]$ với một u. Ta chứng minh phương trình (34.6) bằng phương pháp quy nạp trên u. Với $u = 0$, ta có $i = q$, và biện luận là đúng bởi $P_q \sqsupset P_q$. Dùng hệ thức $P_{\pi[i]} \sqsupset P_i$

i	1	2	3	4	5	6	7	8	9	10
$P[i]$	a	b	a	b	a	b	a	b	c	a
$\pi[i]$	0	0	1	2	3	4	5	6	0	1

(a)



(b)

Hình 34.10 Một minh họa của Bổ đề 34.5 với khuôn mẫu P = ababababca và q = 8. **(a)** Hàm n cho khuôn mẫu đã cho. Bởi $\pi[8] = 6$, $\pi[6] = 4$, $\pi[4] = 2$, và $\pi[2] = 0$, bằng cách lặp lại π ta được $\pi^*[8] = \{8, 6, 4, 2, 0\}$. **(b)** Ta trượt tập mẫu chứa khuôn mẫu P về bên phải và lưu ý khi vài tiền tố P_k của P so khớp với vài hậu tố riêng của P_8; điều này xảy ra với k = 6, 4, 2, và 0. Trong hình, hàng đầu tiên cho ra P và vạch dọc chấm cách được vẽ ngay sau P_8. Các hàng sau đó nêu tất cả các khóa chuyển của P khiến vài tiền tố P_k của P so khớp với vài hậu tố của P_8. Các ký tự so khớp thành công được tô bóng. Các vạch dọc nối các ký tự so khớp được căn thẳng hàng. Như vậy, $\{k : P_k \sqsupset P_q\} = \{8, 6, 4, 2, 0\}$. Bổ đề cho rằng $\pi^*[q] = \{k : P_k \sqsupset P_q\}$ với tất cả q.

và tính bắc cầu của $\sqsupset$ thiết lập biện luận với tất cả i' trong $\pi^*[q]$. Do đó, $\pi^*[q] \subseteq \{k : P_k \sqsupset P_q\}$.

Ta chứng minh rằng $\{k : P_k \quad P_q\} \subseteq \pi^*[q]$ theo sự mâu thuẫn. Giả sử trái ngược lại rằng có một số nguyên trong tập hợp $\{k : P_k \sqsupset P_q\} - \pi^*[q]$, và cho j là giá trị lớn nhất như vậy. Bởi q nằm trong $\{k : P_k \sqsupset P_q\} \cap \pi^*[q]$, ta có $j < q$, và do đó ta cho j' thể hiện số nguyên nhỏ nhất trong $\pi^*[q]$ lớn hơn j. (Ta có thể chọn $j' = q$ nếu không có số khác trong $\pi^*[q]$ lớn hơn j.) Ta có $P_j \sqsupset P_q$ bởi $j \in \{k : P_k \sqsupset P_q\}$, $P_{j'} \sqsupset P_q$ bởi $j' \in \pi^*[q]$; như vậy, $P_j \sqsupset P_{j'}$ theo Bổ đề 34.1. Hơn nữa, j là giá trị lớn nhất như vậy có tính chất này. Do đó, ta phải có $\pi[j'] = j$ và như vậy $j \in \pi^*[q]$. Sự mâu thuẫn này chứng minh bổ đề.

Hình 34.10 minh họa bổ đề này.

Thuật toán COMPUTE-PREFIX-FUNCTION tính toán $\pi[q]$ theo thứ

tự với $q = 1, 2, ..., m$. Phép tính của $\pi[1] = 0$ trong dòng 2 của COMPUTE-PREFIX-FUNCTION chắc chắn là đúng, bởi $\pi[q] < q$ với tất cả q. Bổ đề dưới đây và hệ luận của nó sẽ được dùng để chứng minh COMPUTE-PREFIX-FUNCTION tính toán $\pi[q]$ đúng đắn với $q > 1$.

Bổ đề 34.6

Cho P là một khuôn mẫu có chiều dài m, và cho π là hàm tiền tố với P. Với $q = 1, 2, ..., m$, nếu $\pi[q] > 0$, thì $\pi[q] - 1 \in \pi^*[q-1]$.

Chứng minh Nếu $k = \pi[q] > 0$, thì $P_k \sqsupset P_q$, và như vậy $P_{k-1} \sqsupset P_{q-1}$ (bằng cách thải bỏ ký tự cuối ra khỏi P_k và P_q). Do đó, theo Bổ đề 34.5, $k - 1 \in \pi^*[q-1]$.

Với $q = 2, 3, ..., m$, hãy định nghĩa tập hợp con $E_{q-1} \subseteq \pi^*[q-1]$ theo

$$E_{q-1} = \{k : k \in \pi^*[q-1] \text{ và } P[k+1] = P[q]\} .$$

Tập hợp E_{q-1} bao gồm các giá trị k mà $P_k \sqsupset P_{q-1}$ (theo Bổ đề 34.5); bởi vì $P[k+1] = P[q]$, nó cũng là trường hợp mà với các giá trị này của k, $P_{k+1} \sqsupset P_q$. Theo trực giác, E_{q-1} bao gồm các giá trị $k \in \pi^*[q-1]$ sao cho ta có thể mở rộng P_k đến P_{k+1} và có một hậu tố của P_q.

Hệ luận 34.7

Cho P là một khuôn mẫu có chiều dài m, và cho π là hàm tiền tố với P. Với $q = 2, 3, ..., m$,

$$\pi[q] = \begin{cases} 0 & \text{nếu } E_{q-1} = \phi , \\ 1 + \max\{k \in E_{q-1}\} & \text{nếu } E_{q-1} \neq \phi . \end{cases}$$

Chứng minh Nếu $r = \pi[q]$, thì $P_r \sqsupset P_q$, và do đó $r \geq 1$ hàm ý $P[r] = P[q]$. Do đó, theo Bổ đề 34.6, nếu $r \geq 1$, thì

$$r = 1 + \max\{k \in \pi^*[q-1] : P[k+1] = P[q]\} .$$

Nhưng tập hợp được tối đa hóa chính là E_{q-1}, sao cho $r = 1 + \max\{k \in E_{q-1}\}$ và E_{q-1} không trống. Nếu $r = 0$, sẽ không có $k \in \pi^*[q-1]$ mà ta có thể mở rộng P_k đến P_{k+1} và có một hậu tố của P_q, từ đó ta sẽ có $\pi[q] > 0$. Như vậy, $E_{q-1} = \phi$.

Giờ đây, ta hoàn tất phần chứng minh rằng COMPUTE-PREFIX-FUNCTION tính toán π đúng đắn. Trong thủ tục COMPUTE-PREFIX-FUNCTION, vào đầu mỗi lần lặp lại của vòng lặp **for** trong các dòng 4-9, ta có $k = \pi[q-1]$. Điều kiện này được áp đặt bởi các dòng 2 và 3 khi nhập vòng lặp lần đầu tiên, và nó vẫn đúng trong mỗi lần lặp lại sau đó bởi dòng 9. Các dòng 5-8 điều chỉnh k sao cho giờ đây nó trở thành giá trị đúng đắn của $\pi[q]$. Vòng lặp trên các dòng 5-6 tìm qua tất cả các giá trị $k \in \pi^*[q-1]$ cho đến khi tìm thấy một mà $P[k+1] = P[q]$; tại điểm đó, ta có k là giá trị lớn nhất trong tập hợp E_{q-1}, sao cho, theo Hệ luận 34.7,

ta có thể ấn định $\pi[q]$ theo $k + 1$. Nếu không tìm thấy một k như vậy, $k = 0$ trong các dòng 7-9, và $\pi[q]$ được ấn định là 0. Điều này hoàn tất chứng minh của chúng ta về tính đúng đắn của COMPUTE-PREFIX-FUNCTION.

Tính đúng đắn của thuật toán KMP

Thủ tục KMP-MATCHER có thể được xem là một dạng thực thi lại của thủ tục FINITE-AUTOMATON-MATCHER. Cụ thể, ta sẽ chứng minh rằng mã trên các dòng 6-9 của KMP-MATCHER tương đương với dòng 4 của FINITE-AUTOMATON-MATCHER, ấn định q theo $\delta(q, T[i])$. Tuy nhiên, thay vì dùng một giá trị đã lưu trữ của $\delta(q, T[i])$, giá trị này được tính toán lại từ π khi cần. Sau khi ta đã chứng minh KMP-MATCHER mô phỏng cách ứng xử của FINITE-AUTOMATON-MATCHER, tính đúng đắn của KMP-MATCHER là do tính đúng đắn của FINITE-AUTOMATON-MATCHER (tuy ta sẽ thấy tại sao dòng 12 trong KMP-MATCHER lại cần thiết).

Tính đúng đắn của KMP-MATCHER là do biện luận rằng $\delta(q, T[i]) = 0$ nếu không $\delta(q, T[i]) - 1 \in \pi^*[q]$. Để kiểm tra biện luận này, cho $k = \delta(q, T[i])$. Như vậy, $P_k \sqsupset P_q T[i]$ theo các phần định nghĩa của δ và σ. Do đó, hoặc $k = 0$ hoặc bằng không $k \geq 1$ và $P_{k-1} \sqsupset P_q$ bằng cách bỏ ký tự cuối ra khỏi cả P_k lẫn $P_q T[i]$ (trong trường hợp đó $k - 1 \in \pi^*[q]$). Do đó, hoặc $k = 0$ hoặc $k - 1 \in \pi^*[q]$, chứng minh biện luận.

Biện luận được dùng như sau. Cho q' thể hiện giá trị của q khi dòng 6 được nhập. Ta dùng sự tương đương $\pi^*[q] = \{k : P_k \sqsupset P_q\}$ để xác minh lần lặp lại $q \leftarrow \pi[q]$ điểm danh các thành phần của $\{k : P_k \sqsupset P_q\}$. Các dòng 6-9 xác định $\delta(q', T[i])$ bằng cách xét các thành phần của $\pi^*[q']$ theo thứ tự giảm. Mã sử dụng biện luận để bắt đầu với $q = \phi(T_{i-1}) = \sigma(T_{i-1})$ và thực hiện lần lặp lại $q \leftarrow \pi[q]$ cho đến khi tìm thấy một q sao cho $q = 0$ hoặc $P[q + 1] = T[i]$. Trong trường hợp trước, $\delta(q', T[i]) = 0$; trong trường hợp sau, q là thành phần cực đại trong E_q, sao cho $\delta(q', T[i]) = q + 1$ theo Hệ luận 34.7.

Dòng 12 là cần thiết trong KMP-MATCHER để tránh một tham chiếu khả dĩ đến

$P[m+1]$ trên dòng 6 sau khi tìm thấy một trường hợp xuất hiện của P. (Lập luận rằng $q = \sigma(T_{i-1})$ vào lần thi hành kế tiếp của dòng 6 vẫn hợp lệ theo lời mách nước trong Bài tập 34.4-6: $\delta(m, a) = \delta(\pi[m], a)$ hoặc, theo tương đương, $\sigma(Pa) = \sigma(P_{\pi[m]}a)$ với bất kỳ $a \in \Sigma$) Phần chứng minh còn lại về tính đúng đắn của thuật toán Knuth-Morris-Pratt là do tính đúng đắn của FINITE-AUTOMATON-MATCHER, bởi giờ đây KMP-MATCHER mô phỏng cách ứng xử của FINITE-AUTOMATON-MATCHER.

Bài tập

34.4-1

Tính toán hàm tiền tố π với khuôn mẫu ababbabbababbababbabb khi bảng chữ cái là $\Sigma = \{a, b\}$.

34.4-2

Nêu một cận trên trên kích cỡ của $\pi^*[q]$ dưới dạng một hàm của q. Nêu một ví dụ để chứng tỏ cận của bạn là chặt.

34.4-3

Giải thích cách xác định các lần xuất hiện của khuôn mẫu P trong văn bản T bằng cách xét hàm π với chuỗi PT (chuỗi có chiều dài $m + n$ là phép ghép nối của P và T).

34.4-4

Nêu cách cải thiện KMP-MATCHER bằng cách thay lần xuất hiện của π trong dòng 7 (nhưng không phải dòng 12) bằng π', ở đó π' được định nghĩa một cách đệ quy với $q = 1, 2, \ldots, m$ theo phương trình

$$\pi'[q] = \begin{cases} 0 & \text{nếu } \pi[q] = 0 , \\ \pi'[\pi[q]] & \text{nếu } \pi[q] \neq 0 \text{ và } P[\pi[q] + 1] = P[q + 1] , \\ \pi[q] & \text{nếu } \pi[q] \neq 0 \text{ và } P[\pi[q + 1] \neq P[q + 1] . \end{cases}$$

Giải thích tại sao thuật toán đã sửa đổi lại đúng, và giải thích kiểu sửa đổi này có ý nghĩa gì về mặt cải tiến.

34.4-5

Nêu một thuật toán thời gian tuyến tính để xác định xem một văn bản T có phải là một phép quay chu trình của một chuỗi T' khác hay không. Ví dụ, arc và car là các phép quay chu trình của nhau.

*34.4-6 **

Nêu một thuật toán hiệu quả để tính toán hàm chuyển tiếp δ cho otomat so khớp chuỗi tương ứng với một khuôn mẫu P đã cho. Thuật toán của bạn phải chạy trong thời gian $O(m|\Sigma|)$. (*Mách nước*: Chứng minh rằng $\delta(q, a) = \delta(\pi[q], a)$ nếu $q = m$ hoặc $P[q + 1] \neq a$)

* 34.5 Thuật toán Boyer-Moore

Nếu khuôn mẫu P tương đối dài và bảng chữ cái Σ lớn một cách hợp lý, thì thuật toán so khớp chuỗi hiệu quả nhất ắt là thuật toán của Robert S. Boyer và J. Strother Moore.

```
BOYER-MOORE-MATCHER(T, P, Σ)
1   n ← length[T]
2   m ← length[P]
3   λ ← COMPUTE-LAST-OCCURRENCE-FUNCTION(P, m, Σ)
4   γ ← COMPUTE-GOOD-SUFFIX-FUNCTION(P, m)
5   s ← 0
6   while s ≤ n − m
7        do j ← m
8           while j > 0 và P[j] = T[s + j]
9               do j ← j − 1
10          if j = 0
11             then in "Pattern occurs at shift" s
12                  s ← s + γ[0]
13             else s ← s + max(γ[j], j − λ[T[s + j]])
```

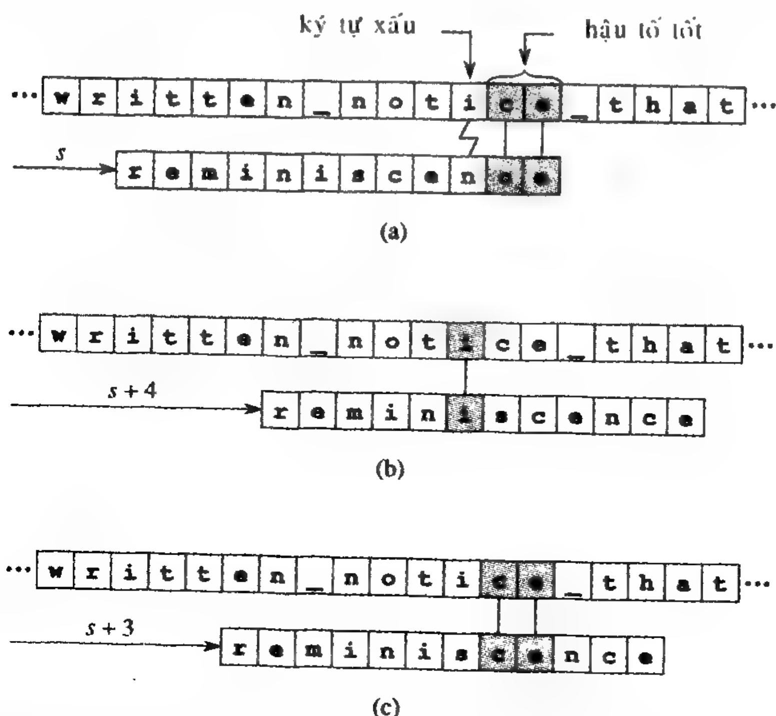
Ngoài các λ và γ trông có vẻ bí ẩn, chương trình này rất giống với thuật toán so khớp chuỗi đơn sơ. Quả vậy, giả sử ta đánh dấu chú giải các dòng 3-4 và thay đợt cập nhật của s trên các dòng 12-13 bằng các phép gia số đơn giản như sau:

```
12                  s ← s + 1
13             else s ← s + 1
```

Chương trình đã sửa đổi giờ đây tác động y hệt như bộ so khớp chuỗi đơn sơ: vòng lặp **while** bắt đầu trên dòng 6 lần lượt xét từng trong số $n - m + 1$ khóa chuyển khả dĩ s, và vòng lặp **while** bắt đầu trên dòng 8 sẽ kiểm tra điều kiện $P[1..m] = T[s + 1..s + m]$ bằng cách so sánh $P[j]$ với $T[s + j]$ với $j = m, m - 1, \ldots, 1$. Nếu vòng lặp kết thúc bằng $j = 0$, một khóa chuyển hợp lệ s đã được tìm thấy, và dòng 11 in ra giá trị của s. Tại cấp này, các tính năng đặc biệt duy nhất của thuật toán Boyer-Moore đó là nó so sánh khuôn mẫu đối với văn bản *từ phải qua trái* và nó gia tăng khóa chuyển s trên các dòng 12-13 theo một giá trị không nhất thiết là 1.

Thuật toán Boyer-Moore liên kết hai heuristic cho phép nó tránh được phần lớn công việc mà các thuật toán so khớp chuỗi trước đây của chúng ta đã thực hiện. Các heuristic này hiệu quả đến nỗi chúng thường cho phép thuật toán bỏ qua việc xem xét nhiều ký tự văn bản. Các heuristic này, được mệnh danh là "heuristic ký tự xấu" và "heuristic hậu tố tốt," được minh họa trong Hình 34.11. Chúng có thể được xem là

hoạt động song song độc lập. Khi xảy ra một trường hợp không so khớp, mỗi heuristic đề xuất một lượng qua đó s có thể an toàn gia tăng mà không mất đi một khóa chuyển hợp lệ. Thuật toán Boyer-Moore chọn lượng lớn hơn và gia tăng s theo lượng đó: khi đạt đến dòng 13 sau một trường hợp không so khớp, heuristic ký tự xấu đề xuất tăng s lên $j - \lambda[T[s + j]]$, và heuristic hậu tố tốt đề xuất tăng s lên $\gamma[j]$.



Hình 34.11 Một minh họa của các heuristic Boyer. **(a)** So khớp phần hồi tương khuôn mẫu đối lại một văn bản bằng cách so sánh các ký tự theo cách từ phải qua trái. Khóa chuyển s không hợp lệ: mặc dù một "hậu tố tốt" ce của khuôn mẫu được so khớp đúng đắn đối lại các ký tự tương ứng trong văn bản (các ký tự so khớp được nêu ở dạng tô bóng). "ký tự xấu" i, không so khớp ký tự tương ứng n trong khuôn mẫu, đã được khám phá trong văn bản. **(b)** Heuristic ký tự xấu đề xuất dời khuôn mẫu sang phải, nếu được, theo lượng bảo đảm ký tự văn bản xấu sẽ so khớp trường hợp xuất hiện mút phải của ký tự xấu trong khuôn mẫu. Trong ví dụ này, việc dời khuôn mẫu sang phải 4 vị trí sẽ khiến ký tự văn bản xấu i trong văn bản so khớp i mút phải trong khuôn mẫu, tại vị trí 6. Nếu ký tự xấu không xảy ra trong khuôn mẫu, thì khuôn mẫu có thể được dời hoàn toàn quá ký tự xấu trong văn bản. Nếu trường hợp xuất hiện mút phải của ký tự xấu trong khuôn mẫu nằm về bên phải của vị trí ký tự xấu hiện hành, thì heuristic này không đề xuất gì cả. **(c)** Với heuristic hậu tố tốt, khuôn mẫu được dời sang phải theo lượng bé nhất bảo đảm mọi ký tự khuôn mẫu căn thẳng hàng với hậu tố tốt ce đã tìm thấy trước đó trong văn bản sẽ so khớp các ký tự hậu tố đó. Trong ví dụ này, việc dời khuôn mẫu sang phải 3 vị trí sẽ thỏa điều kiện này. Bởi heuristic hậu tố tốt đề xuất một đợt dời 3 vị trí, nhỏ hơn đề xuất 4 vị trí của heuristic ký tự xấu, thuật toán Boyer-Moore gia tăng khóa chuyển lên 4.

Heuristic ký tự xấu

Khi xảy ra một trường hợp không so khớp, heuristic ký tự xấu sử dụng thông tin về nơi ký tự văn bản xấu $T[s + j]$ xảy ra trong khuôn mẫu (nếu có nó xảy ra) để đề xuất một khóa chuyển mới. Trong trường hợp tốt nhất, trường hợp không so khớp xảy ra trên phép so sánh đầu tiên ($P[m] \neq T[s + m]$) và ký tự xấu $T[s + m]$ không xảy ra trong khuôn mẫu gì cả. (Hãy tưởng tượng tìm kiếm a^m trong chuỗi văn bản b^n.) Trong trường hợp này, ta có thể gia tăng khóa chuyển s lên m, bởi mọi khóa chuyển nhỏ hơn $s + m$ sẽ căn thẳng hàng một ký tự khuôn mẫu nào đó đối lại ký tự xấu, gây ra một trường hợp không so khớp. Nếu trường hợp tốt nhất xảy ra liên tục, thuật toán Boyer-Moore chỉ xét một phân số $1/m$ ký tự văn bản, bởi mỗi ký tự văn bản được xem xét cho ra một trường hợp không so khớp, như vậy khiến s tăng lên m. Cách ứng xử trường hợp tốt nhất này minh họa năng lực của tính năng so khớp phải qua trái thay vì trái qua phải.

Nói chung, ***heuristic ký tự xấu*** làm việc như sau. Giả sử ta vừa tìm thấy một trường hợp không so khớp: $P[j] \neq T[s + j]$ với một j, ở đó $1 \leq j \leq m$. Như vậy, ta cho k là chỉ số lớn nhất trong miền giá trị $1 \leq k \leq m$ sao cho $T[s + j] = P[k]$, nếu có một k bất kỳ như vậy tồn tại. Bằng không, ta cho $k = 0$. Ta biện luận rằng có thể an toàn gia tăng s lên $j - k$. Ta phải xét ba trường hợp để chứng minh biện luận này, như minh họa trong Hình 34.12.

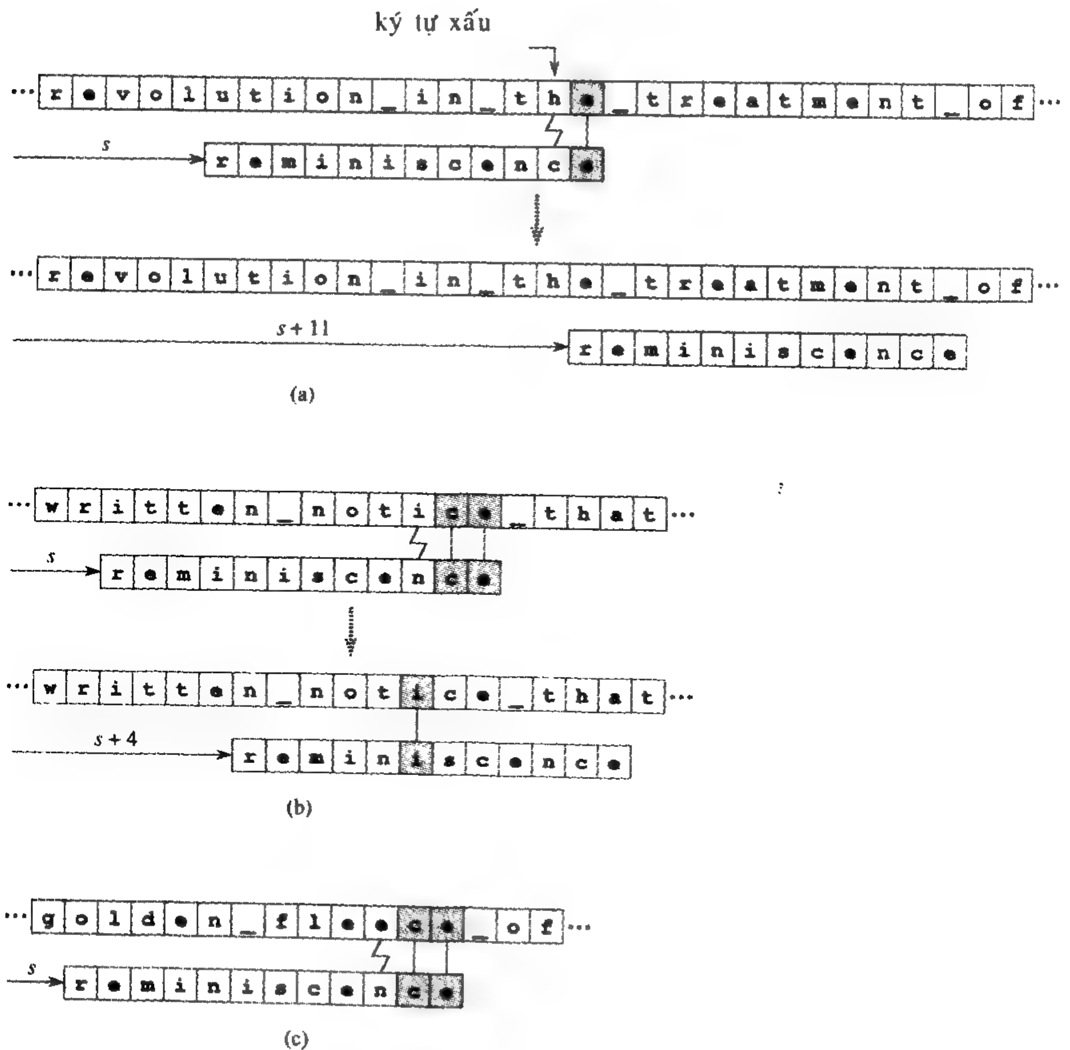
- $k = 0$: Như đã nêu trong Hình 34.12(a), ký tự xấu $T[s + j]$ không xảy ra trong khuôn mẫu gì cả, và do đó ta có thể an toàn gia tăng s lên j mà không làm mất các khóa chuyển hợp lệ.

- $k < j$: Như đã nêu trong Hình 34.12(b), trường hợp xuất hiện mút phải của ký tự xấu nằm trong khuôn mẫu về bên trái của vị trí j, sao cho $j - k > 0$ và khuôn mẫu phải được dời $j - k$ ký tự về bên phải trước khi ký tự văn bản xấu so khớp một ký tự khuôn mẫu bất kỳ. Do đó, ta có thể an toàn gia tăng s lên $j - k$ mà không làm mất các khóa chuyển hợp lệ.

- $k > j$: Như đã nêu trong Hình 34.12(c), $j - k < 0$, và do đó về cơ bản heuristic ký tự xấu đang đề xuất giảm s. Lời đề nghị này sẽ được thuật toán Boyer-Moore bỏ qua, bởi heuristic hậu tố tốt sẽ đề xuất một khóa chuyển về bên phải trong tất cả các trường hợp.

Chương trình đơn giản dưới đây định nghĩa $\lambda[a]$ là chỉ số của vị trí mút phải trong khuôn mẫu nơi ký tự a xảy ra, với mỗi $a \in \Sigma$. Nếu a không xảy ra trong khuôn mẫu, thì $\lambda[a]$ được ấn định là 0. Ta gọi λ là *hàm trường hợp xuất hiện chót* [last-occurrence function] cho khuôn mẫu. Với định nghĩa này, biểu thức $j - \lambda[T[s+j]]$ trên dòng 13 của BOYER-MOORE MATCHER thực thi heuristic ký tự xấu. (Bởi $j - \lambda[T[s + j]]$ là

âm nếu trường hợp xuất hiện mút phải của ký tự xấu $T[s + j]$ trong khuôn mẫu nằm về bên phải của vị trí j, ta dựa vào tính xác thực của $\gamma[j]$, mà heuristic hậu tố tốt đề xuất, để bảo đảm thuật toán tiến triển tại mỗi bước.)



Hình 34.12 Các trường hợp của heuristic ký tự xấu. **(a)** Ký tự xấu h không xảy ra ở đâu cả trong khuôn mẫu, và do đó khuôn mẫu có thể được đẩy tới $j = 11$ ký tự cho đến khi nó chuyển qua ký tự xấu. **(b)** Trường hợp xuất hiện mút phải của ký tự xấu trong khuôn mẫu nằm tại vị trí $k < j$, và do đó khuôn mẫu có thể được đẩy tới $j - k$ ký tự. Bởi $j = 10$ và $k = 6$ với ký tự xấu i, nên khuôn mẫu có thể được đẩy tới 4 vị trí cho đến khi các i thẳng hàng. **(c)** Trường hợp xuất hiện mút phải của ký tự xấu trong khuôn mẫu nằm tại vị trí $k > j$. Trong ví dụ này, $j = 10$ và $k = 12$ với ký tự xấu e. Heuristic ký tự xấu đề xuất một khóa chuyển âm, được bỏ qua.

```
COMPUTE-LAST-OCCURRENCE-FUNCTION(P, m, Σ)
for mỗi ký tự a ∈ Σ
do λ[a] = 0
for j ← 1 to m
do λ[P[j]] ← j
return λ
```

Thời gian thực hiện của thủ tục COMPUTE-LAST-OCCURRENCE-FUNCTION là $O(|\Sigma| + m)$.

Heuristic hậu tố tốt

Ta hãy định nghĩa hệ thức $Q \sim R$ (đọc là "Q tương tự R") với các chuỗi Q và R có nghĩa là $Q \sqsupset R$ hoặc $R \sqsupset Q$. Nếu hai chuỗi tương tự, thì ta có thể căn thẳng hàng chúng với các ký tự mút phải của chúng đã so khớp, và không có cặp ký tự đã căn thẳng hàng nào không khớp. Hệ thức "~" là đối xứng: $Q \sim R$ nếu và chỉ nếu $R \sim Q$. Với tư cách là hệ quả của Bổ đề 34.1, ta cũng có

$$Q \sqsupset R \text{ và } S \sqsupset R \text{ ngụ ý } Q \sim S. \qquad (34.7)$$

Nếu ta thấy rằng $P[j] \neq T[s + j]$, ở đó $j < m$, thì ***heuristic hậu tố tốt*** nói rằng ta có thể an toàn đẩy s tới theo

$$\gamma[j] = m - \max\{k : 0 \leq k < m \text{ and } P[j + 1..m] \sim P_k\}.$$

Nghĩa là, $\gamma[j]$ là lượng bé nhất mà ta có thể đẩy s tới và không gây cho các ký tự trong "hậu tố tốt" $T[s + j + 1..s + m)$ không so khớp đối lại cách căn thẳng hàng mới của khuôn mẫu. Hàm γ được định nghĩa kỹ với tất cả j, bởi $P[j + 1..m] \sim P_0$ với tất cả j: chuỗi trống tương tự với tất cả các chuỗi. Ta gọi γ là ***hàm hậu tố tốt*** với khuôn mẫu P.

Giờ đây, ta nêu cách tính toán hàm hậu tố tốt γ. Trước tiên, ta nhận thấy $\gamma[j] \leq m - \pi[m]$ với tất cả j, như sau. Nếu $w = \pi(m)$, thì $P_w \sqsupset P$ theo định nghĩa của π. Vả lại, bởi $P[j + 1..m] \sqsupset P$ với bất kỳ j, ta có $P_w \sim P[j + 1..m]$, theo phương trình (34.7). Do đó, $\gamma[j] \leq m - \pi[m]$ với tất cả j.

Giờ đây, ta có thể viết lại định nghĩa của γ là

$$\gamma[j] = m - \max\{k : \pi[m] \leq k < m \text{ và } P[j + 1..m] \sim P_k\}.$$

Điều kiện $P[j + 1..m] \sim P_k$ sẽ đứng vững nếu hoặc $P[j + 1..m] \sqsupset P_k$ hoặc $P_k \sqsupset P[j + 1..m]$. Nhưng khả năng thứ hai này hàm ý rằng $P_k \sqsupset P$ và như vậy $k \leq \pi[m]$, theo định nghĩa của π. Khả năng thứ hai này không thể rút gọn giá trị của $\gamma[j]$ bên dưới $m - \pi[m]$. Do đó, ta vẫn có thể viết lại thêm định nghĩa của γ như sau:

$$\gamma[j] = m - \max(\{\pi[m]\} \cup \{k : \pi[m] < k < m \text{ và } P[j + 1..m] \sqsupset P_k\}).$$

(Tập hợp thứ hai có thể trống.) Cũng đáng để nhận xét rằng phần định nghĩa hàm ý rằng $\gamma[j] > 0$ với tất cả $j = 1, 2, \ldots, m$, điều này bảo đảm thuật toán Boyer-Moore tiến triển.

Để rút gọn hơn nữa biểu thức cho γ, ta định nghĩa P' là nghịch đảo của khuôn mẫu P và π' là hàm tiền tố tương ứng. Nghĩa là, $P'[i] = P[m - i + 1]$ với $i = 1, 2, \ldots, m$, và $\pi'[t]$ là u lớn nhất sao cho $u < t$ và $P'_u \sqsupset P'_t$.

Nếu k là giá trị khả dĩ lớn nhất sao cho $P[j + 1..m] \sqsupset P_k$, thì ta biện luận rằng

$$\pi'[l] = m - j \qquad (34.8)$$

ở đó $l = (m - k) + (m - j)$. Để xem biện luận này được định nghĩa kỹ, ta lưu ý $P[j + 1..m] \sqsupset P_k$ hàm ý rằng $m - j \le k$, và như vậy $l \le m$. Ngoài ra, $j < m$ và $k \le m$, sao cho $l \ge 1$. Ta chứng minh biện luận này như sau. Bởi $P[j + 1..m] \sqsupset P_k$, ta có $P'_{m-j} \sqsupset P'_l$. Do đó, $\pi'[l] \ge m - j$. Giờ đây, giả sử $p > m - j$, ở đó $p = \pi'[l]$. Như vậy, theo định nghĩa của π', ta có $P'_p \sqsupset P'_l$ hoặc, theo tương đương, $P'[1..p] = P'[l - p + 1..l]$. Viết lại phương trình này theo dạng P thay vì P', ta có $P[m - p + 1..m] = P[m - l + 1..m - l + p]$. Thay $l = 2m - k - j$, ta được $P[m - p + 1..m] = P[k - m + j + 1..k - m + j + p]$, hàm ý $P[m - p + 1..m] \sqsupset P_{k-m+j+p}$. Bởi $p > m - j$, ta có $j + 1 > m - p + 1$, và do đó $P[j + 1..m] \sqsupset P[m - p + 1..m]$, hàm ý $P[j + 1..m] \sqsupset P_{k-m+j+p}$ theo tính bắc cầu của $\sqsupset$. Cuối cùng, bởi $p > m - j$, ta có $k' > k$, ở đó $k' = k - m + j + p$, mâu thuẫn với chọn lựa k của chúng ta dưới dạng giá trị khả dĩ lớn nhất sao cho $P[j + 1..m] \sqsupset P_k$. Sự mâu thuẫn này có nghĩa là ta không thể có $p > m - j$, và như vậy $p = m - j$, chứng minh biện luận (34.8).

Dùng phương trình (34.8), và lưu ý rằng $\pi'[l] = m - j$ hàm ý rằng $j = m - \pi'[l]$ và $k = m - l + \pi'[l]$, ta vẫn có thể viết lại thêm định nghĩa của γ:

$$\begin{aligned}
\gamma[j] &= m - \max(\{\pi[m]\} \\
&\qquad \cup \{m - l + \pi'[l] : 1 \le l \le m \text{ và } j = m - \pi'[l]\}) \\
&= \min(\{m - \pi[m]\} \\
&\qquad \cup \{l - \pi'[l] : 1 \le l \le m \text{ và } j = m - \pi'[l]\}) . \qquad (34.9)
\end{aligned}$$

Một lần nữa, tập hợp thứ hai có thể trống.

Giờ đây ta sẵn sàng xét thủ tục để tính toán γ.

```
COMPUTE-GOOD-SUFFIX-FUNCTION(P, m)
    π ← COMPUTE-PREFIX-FUNCTION(P)
    P' ← reverse(P)
    π' ← COMPUTE-PREFIX-FUNCTION(P')
```

```
for j ← 0 to m
        do γ[j] ← m − π'[m]
for l ← 1 to m
        do j ← m − π'[l]
            if γ[j] > l − π'[l]
                then γ[j] ← l − π'[l]
return γ
```

Thủ tục COMPUTE-GOOD-SUFFIX-FUNCTION là một thực thi đơn giản của phương trình (34.9). Thời gian thực hiện của nó là $O(m)$.

Thời gian thực hiện trường hợp xấu nhất của thuật toán Boyer-Moore rõ ràng là $O((n-m+1)m+|\Sigma|)$, bởi COMPUTE-LAST-OCCURRENCE-FUNCTION mất một thời gian $O(m+|\Sigma|)$, COMPUTE-GOOD-SUFFIX-FUNCTION mất một thời gian $O(m)$, và thuật toán Boyer-Moore (giống như thuật toán Rabin-Karp) mất $O(m)$ thời gian phê chuẩn mỗi khóa chuyển hợp lệ s. Tuy nhiên, trong thực tế, thuật toán Boyer-Moore thường là thuật toán được chọn lựa.

Bài tập

34.5-1

Tính toán các hàm λ và γ cho khuôn mẫu P = 0101101201 và bảng chữ cái $\Sigma = \{0, 1, 2\}$.

34.5-2

Nêu các ví dụ để chứng tỏ rằng nhờ tổ hợp heuristic ký tự xấu và heuristic hậu tố tốt, thuật toán Boyer-Moore có thể thực hiện tốt hơn nhiều so với trường hợp chỉ dùng heuristic hậu tố tốt.

34.5-3 *

Một cải tiến cho thủ tục Boyer-Moore cơ bản thường được dùng trong thực tế đó là thay hàm γ bằng γ', được định nghĩa bởi

$$\gamma'[j] = m - \max\{k : 0 \le k < m \text{ and } P[j+1..m] \sim P_k \text{ và}$$
$$(k - m + j > 0 \text{ hàm ý } P[j] \ne P[k-m+j])\}\ .$$

Ngoài việc bảo đảm các ký tự trong hậu tố tốt sẽ không so khớp tại khóa chuyển mới, hàm γ' còn bảo đảm cùng ký tự khuôn mẫu sẽ không so khớp tiếp đối lại ký tự văn bản xấu. Nêu cách tính toán hàm γ' một cách hiệu quả.

Các Bài Toán

34-1 So khớp chuỗi dựa trên các thừa số lặp lại

Cho y^i thể hiện phép ghép nối chuỗi y với chính nó i lần. Ví dụ, $(ab)^3 = ababab$. Ta nói rằng một chuỗi $x \in \Sigma^*$ có ***thừa số lặp lại*** r nếu $x = y^r$ với một chuỗi $y \in \Sigma^*$ và một $r > 0$. Cho $\rho(x)$ thể hiện r lớn nhất sao cho x có thừa số lặp lại r.

a. Nêu một thuật toán hiệu quả nhận một khuôn mẫu $P[1..m]$ làm đầu vào và tính toán $\rho(P_i)$ với $i = 1, 2, ..., m$. Nêu thời gian thực hiện của thuật toán?

b. Với bất kỳ khuôn mẫu $P[1..m]$, cho $\rho^*(P)$ được định nghĩa là $\max_{1 \le i \le m} \rho(P_i)$. Chứng minh rằng nếu khuôn mẫu P được chọn ngẫu nhiên từ tập hợp tất cả các chuỗi nhị phân có chiều dài m, thì giá trị dự trù của $\rho^*(P)$ là $O(1)$.

c. Chứng tỏ thuật toán so khớp chuỗi dưới đây tìm đúng đắn tất cả các lần xuất hiện của khuôn mẫu P trong một văn bản $T[1..n]$ trong thời gian $O(\rho^*(P)n + m)$.

```
REPETITION-MATCHER(P, T)
m ← length[P]
n ← length[T]
k ← 1 + ρ*(P)
q ← 0
s ← 0
while s ≤ n − m
      do if T[s + q + 1] = P[q + 1]
            then q ← q + 1
                 if q = m
                    then print "Pattern occurs with shift" s
         if q = m hoặc T[s+q+1] ≠ P[q+1]
            then s ← s + max(1, ⌈q / k⌉)
                 q ← 0
```

Thuật toán này là của Galil và Seiferas. Bằng cách mở rộng đáng kể các ý tưởng này, họ có được một thuật toán so khớp chuỗi thời gian tuyến tính chỉ sử dụng $O(1)$ kho lưu trữ vượt quá mức yêu cầu cho P và T.

34-2 So khớp chuỗi song song

Xét bài toán so khớp chuỗi trên một máy tính song song. Giả sử với một khuôn mẫu đã cho, ta có một otomat so khớp chuỗi M với tập hợp trạng thái Q. Cho ϕ là hàm trạng thái chung cuộc của M. Giả sử rằng văn bản đầu vào của chúng ta là $T[1..n]$. Ta muốn tính toán $\phi(T_i)$ với $i = 1, 2, ..., n$; nghĩa là, ta muốn tính toán trạng thái chung cuộc cho mỗi tiền tố. Chiến lược của chúng ta đó là sử dụng phép tính tiền tố song song mô tả trong Đoạn 30.1.2.

Với bất kỳ chuỗi đầu vào x, định nghĩa hàm $\delta_x : Q \rightarrow Q$ sao cho nếu M bắt đầu trong trạng thái q và đọc đầu vào x, thì M kết thúc trong trạng thái $\delta_x(q)$.

a. Chứng minh $\delta_y \circ \delta_x = \delta_{xy}$, ở đó $\circ$ thể hiện sự cấu tạo chức năng:

$(\delta_y \circ \delta_x)(q) = \delta_x(\delta_y(q))$.

b. Chứng tỏ $\circ$ là một phép toán kết hợp.

c. Chứng tỏ δ_{xy} có thể được tính toán từ các phép biểu diễn bảng biểu của δ_x và δ_y trong $O(1)$ thời gian trên một PRAM CREW. Phân tích xem xần bao nhiêu bộ xử lý theo dạng $|Q|$.

d. Chứng minh $\phi(T_i) = \delta_{T_i}(q_0)$, ở đó q_0 là trạng thái đầu cho M.

e. Nêu cách tìm tất cả các lần xuất hiện của một khuôn mẫu trong một văn bản có chiều dài n trong $O(\lg n)$ thời gian trên một PRAM CREW. Giả sử khuôn mẫu được cung cấp theo dạng otomat so khớp chuỗi tương ứng.

Ghi chú Chương

Quan hệ của so khớp chuỗi với lý thuyết của otomat hữu hạn được Aho, Hopcroft, và Ullman [4] mô tả. Thuật toán Knuth-Morris-Pratt [125] được phát minh độc lập bởi Knuth và Pratt và bởi Morris; họ đã xuất bản công trình hợp tác của họ. Thuật toán Rabin-Karp đã được Rabin và Karp [117] đề xuất, và thuật toán Boyer-Moore là của Boyer và Moore [32]. Galil và Seiferas [78] cho ta một thuật toán so khớp chuỗi thời gian tuyến tính tất định đáng quan tâm chỉ sử dụng $O(1)$ không gian vượt quá mức yêu cầu để lưu trữ khuôn mẫu và văn bản.

35 Hình Học Điện Toán

Hình học điện toán [computational geometry] là một nhánh của khoa học máy tính nghiên cứu các thuật toán để giải các bài toán hình học. Trong toán học và thiết kế kỹ thuật [engineering] hiện đại, hình học điện toán có các ứng dụng về đồ họa máy tính, robot, thiết kế VLSI, thiết kế nhờ máy tính hỗ trợ, thống kê, và nhiều lĩnh vực khác. Nhập liệu cho một bài toán hình học điện toán thường là một mô tả về một tập hợp các đối tượng hình học, như một tập hợp các điểm, một tập hợp các đoạn thẳng, hoặc các đỉnh của một đa giác theo thứ tự ngược chiều kim đồng hồ. Kết xuất thường là một đáp ứng trước một đợt truy vấn về các đối tượng, như có các dòng giao nhau nào không, hoặc giả là một đối tượng hình học mới, như bao lồi (đa giác lồi bao kín nhỏ nhất) của tập hợp các điểm.

Trong chương này, ta xét một vài thuật toán hình học điện toán theo hai chiều, nghĩa là, trong mặt phẳng. Mỗi đối tượng đầu vào được biểu thị dưới dạng một tập hợp các điểm $\{p_i\}$, ở đó mỗi $p_i = <x_i, y_i)$ và $x_i, y_i \in \mathbf{R}$. Ví dụ, đa giác n-đỉnh P được biểu thị bởi một dãy $(p_0, p_1, p_2, ..., p_{n-1}>$ các đỉnh của nó theo thứ tự xuất hiện của chúng trên biên của P. Hình học điện toán cũng có thể được thực hiện theo ba chiều, và thậm chí trong các không gian chiều cao hơn, nhưng những bài toán như vậy và các giải pháp của chúng có thể rất khó hình dung. Tuy nhiên, thậm chí theo hai chiều, ta vẫn có thể xem một mẫu tốt về các kỹ thuật hình học điện toán.

Đoạn 35.1 nêu cách trả lời các câu hỏi đơn giản về các đoạn thẳng một cách hiệu quả và chính xác: một đoạn thẳng sẽ theo chiều kim đồng hồ hay ngược chiều kim đồng hồ từ một đoạn thẳng khác chia sẻ một điểm cuối, ta quay đường nào khi băng ngang hai đoạn thẳng tiếp giáp, và hai đoạn thẳng có giao nhau không. Đoạn 35.2 trình bày một kỹ thuật có tên "phép quét" mà ta dùng để phát triển một thuật toán $O(n \lg n)$ thời gian để xác định có các giao điểm giữa một tập hợp n đoạn thẳng hay không. Đoạn 35.3 cung cấp hai thuật toán "quét quay" tính toán bao lồi (đa giác lồi bao kín nhỏ nhất) của một tập hợp n điểm: quét Graham, chạy trong thời gian $O(n \lg n)$, và diễu hành Jarvis, mất $O(nh)$ thời gian, ở đó h là số lượng các đỉnh của bao lồi. Cuối cùng,

Đoạn 35.4 cung cấp một thuật toán chia để trị $O(n \lg n)$ thời gian để tìm cặp điểm sát nhất trong một tập hợp n điểm trong mặt phẳng.

35.1 Các tính chất đoạn thẳng

Có một số thuật toán hình học điện toán trong chương này sẽ yêu cầu các câu trả lời cho các câu hỏi về các tính chất của các đoạn thẳng. Một ***tổ hợp lồi*** hai điểm riêng biệt $p_1 = (x_1, y_1)$ và $p_2 = (x_2, y_2)$ là bất kỳ điểm $p_3 = (x_3, y_3)$ sao cho với một α trong miền giá trị $0 \leq \alpha \leq 1$, ta có $x_3 = \alpha x_1 + (1 - \alpha)x_2$ và $y_3 = \alpha y_1 + (1 - \alpha)y_2$. Ta cũng viết $p_3 = \alpha p_1 + (1 - \alpha)p_2$. Theo trực giác, p_3 là một điểm bất kỳ nằm trên đường thẳng đi qua p_1 và p_2 và nằm trên hoặc giữa p_1 và p_2 trên đường thẳng. Cho hai điểm riêng biệt p_1 và p_2, ***đoạn thẳng*** $\overline{p_1p_2}$ là tập hợp các tổ hợp lồi của p_1 và p_2. Ta gọi p_1 và p_2 là ***các điểm cuối*** của đoạn thẳng $\overline{p_1p_2}$. Đôi lúc cách sắp xếp thứ tự của p_1 và p_2 có tính chất quan trọng, và ta nói về ***đoạn thẳng có hướng*** [directed segment] $\overrightarrow{p_1p_2}$. Nếu p_1 là ***gốc*** (0, 0), thì ta có thể xem đoạn thẳng có hướng $\overrightarrow{p_1p_2}$ như ***vectơ*** p_2.

Trong đoạn này, ta sẽ khảo sát các câu hỏi sau đây:

1. Cho hai đoạn thẳng có hướng $\overrightarrow{p_0p_1}$ và $\overrightarrow{p_0p_2}$ có phải theo chiều kim đồng hồ từ $\overrightarrow{p_0p_2}$ đối với điểm cuối chung p_0 của chúng hay không?

2. Cho hai đoạn thẳng $\overline{p_1p_2}$ và $\overline{p_2p_3}$, nếu ta băng ngang $\overline{p_1p_2}$ rồi $\overline{p_2p_3}$, ta có quay trái tại điểm p_2 hay không?

3. Các đoạn thẳng $\overline{p_1p_2}$ và $\overline{p_3p_4}$ có giao nhau hay không?

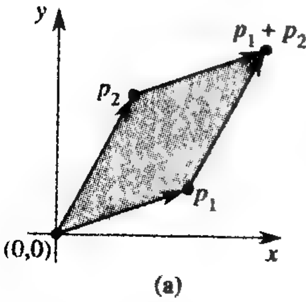
Không có hạn chế nào trên các điểm đã cho.

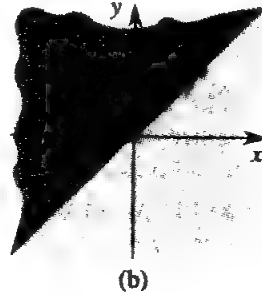
Ta có thể trả lời mỗi câu hỏi trong $O(1)$ thời gian, là điều không có gì đáng ngạc nhiên bởi kích cỡ đầu vào của mỗi câu hỏi là $O(1)$. Hơn nữa, các phương pháp chỉ sử dụng các phép cộng, trừ, nhân, và các phép so sánh. Ta không cần phép chia cũng như các hàm lượng giác, cả hai có thể tốn kém về mặt điện toán và thiên về các bài toán có lỗi làm tròn. Ví dụ, phương pháp "đơn giản" để xác định hai đoạn thẳng giao nhau—tính toán phương trình đường thẳng có dạng $y = mx+b$ cho mỗi đoạn thẳng (m là độ nghiêng và b là đoạn chặn y), tìm điểm giao của các đường thẳng, và kiểm tra xem điểm này có nằm trên cả hai đoạn thẳng hay không—sử dụng phép chia để tìm ra điểm giao. Khi các đoạn thẳng gần song song, phương pháp này rất nhạy cảm với sự chính xác của phép toán chia trên các máy tính thực. Phương pháp trong đoạn này, tránh phép chia, thường chính xác hơn nhiều.

Các tích vectơ

Tính toán các tích vectơ là trọng tâm của các phương pháp đoạn thẳng. Xét các vectơ p_1 và p_2, nêu trong Hình 35. 1 (a). ***Tích vectơ*** $p_1 \times$ pz có thể được diễn dịch dưới dạng vùng có dấu của hình bình hành được hình thành bởi các điểm y các điểm (0, 0), p_1, p_2, và $p_1 + p_2 = (x_1 + x_2, y_1 + y_2)$. Một định nghĩa tương đương, nhưng hữu ích hơn cung cấp tích vectơ dưới dạng định thức của một ma trận[1]

$$p_1 \times p_2 = \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix}$$
$$= x_1 y_2 - x_2 y_1$$
$$= -p_2 \times p_1 .$$





Hình 35.1 (a) Tích vectơ của các vectơ p_1 và p_2 là vùng có dấu của hình bình hành. **(b)** Vùng tô bóng sáng chứa các vectơ theo chiều kim đồng hồ từ p. Vùng tô bóng sẫm chứa các vectơ ngược chiều kim đồng hồ từ p.

Nếu $p_1 \times p_2$ là dương, thì p_1 theo chiều kim đồng hồ từ p_2 đối với gốc (0, 0); nếu tích vectơ này là âm, thì p_1 ngược chiều kim đồng hồ từ p_2. Hình 35.1(b) nêu các vùng theo chiều kim đồng hồ và ngược chiều kim đồng hồ tương đối với một vectơ p. Một điều kiện cận nảy sinh nếu tích vectơ là zero; trong trường hợp này, các vectơ là ***cộng tuyến***, trỏ theo cùng hướng hoặc hướng ngược lại.

Để xác định một đoạn thẳng có hướng $\overrightarrow{p_0 p_1}$ có theo chiều kim đồng hồ từ một đoạn thẳng có hướng $\overrightarrow{p_0 p_2}$ đối với điểm cuối chung p_0 của chúng hay không, ta đơn giản phiên dịch để sử dụng p_0 dưới dạng gốc. Nghĩa là, ta cho $p_1 - p_0$ thể hiện vectơ $p'_1 = (x'_1, y'_1)$, ở đó $x'_1 = x_1 - x_0$ và $y'_1 = y_1 - y_0$, và ta định nghĩa $p_2 - p_0$ tương tự. Như vậy, ta tính toán tích vectơ

---

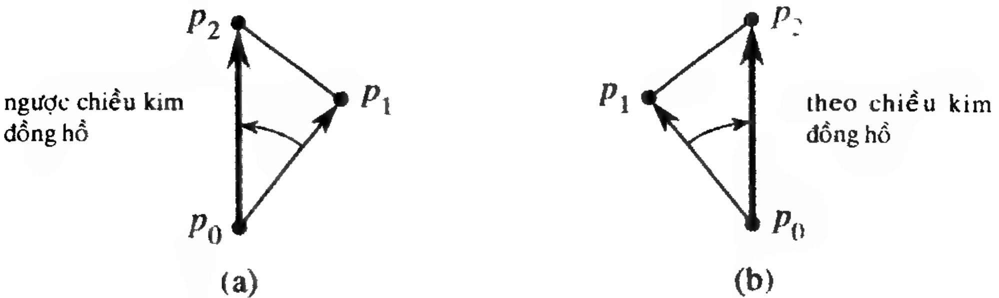
[1] Thực tế, tích vectơ là một khái niệm ba chiều. Nó là một vectơ thẳng góc với cả p_1 lẫn p_2 theo "quy tắc tay phải" và tầm lớn của nó là $|x_1 y_2 - x_2 y_1|$. Tuy nhiên, trong chương này, để tiện dụng, ta xem tích vectơ đơn giản như giá trị $x_1 y_2 - x_2 y_1$

$$(p_1 - p_0) \times (p_2 - p_0) = (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0) .$$

Nếu tích vectơ này là dương, thì $\overrightarrow{p_0 p_1}$ theo chiều kim đồng hồ từ $\overrightarrow{p_0 p_2}$; nếu là âm, thì nó ngược chiều kim đồng hồ.

Xác định các đoạn thẳng liên tiếp quay trái hay phải

Câu hỏi kế tiếp của ta là hai đoạn thẳng liên tiếp $\overline{p_0 p_1}$ và $\overline{p_1 p_2}$ quay trái hay quay phải tại điểm p_1. Theo tương đương, ta muốn có một phương pháp để xác định một góc đã cho $\angle p_0 p_1 p_2$ sẽ quay hướng nào. Các tích vectơ cho phép ta trả lời câu hỏi này mà không tính toán góc. Như đã nêu trong Hình 35.2, ta đơn giản kiểm tra đoạn thẳng có hướng $\overrightarrow{p_0 p_2}$ sẽ theo chiều kim đồng hồ hay ngược chiều kim đồng hồ tương đối với đoạn thẳng có hướng $\overrightarrow{p_0 p_1}$. Để thực hiện, ta tính toán tích vectơ $(p_2 - p_0) \times (p_1 - p_0)$. Nếu dấu của tích vectơ này là âm, thì $\overrightarrow{p_0 p_2}$ ngược chiều kim đồng hồ đối với $\overrightarrow{p_0 p_1}$, và như vậy ta quay trái tại p_1. Một tích vectơ dương nêu rõ một hướng theo chiều kim đồng hồ và quay phải. Một tích vectơ 0 có nghĩa là các điểm p_0, p_1, và p_2 là cộng tuyến.



Hình 35.2 Dùng tích vectơ để xác định cách quay của các đoạn thẳng liên tiếp $\overline{p_0 p_1}$ và $\overline{p_1 p_2}$ tại điểm p_1. Ta kiểm tra đoạn thẳng có hướng $\overrightarrow{p_0 p_2}$ theo chiều kim đồng hồ hay ngược chiều kim đồng hồ tương đối với đoạn thẳng có hướng $\overrightarrow{p_0 p_1}$. **(a)** Nếu ngược chiều kim đồng hồ, các điểm quay trái. **(b)** Nếu theo chiều kim đồng hồ, chúng quay phải.

Xác định hai đoạn thẳng giao nhau

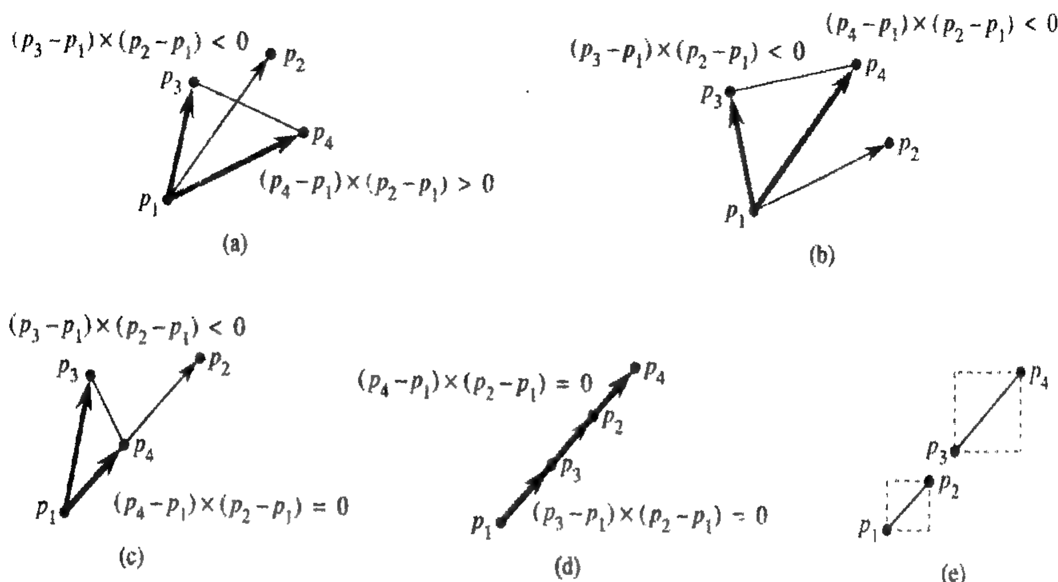
Ta dùng một tiến trình hai giai đoạn để xác định hai đoạn thẳng có giao nhau hay không. Giai đoạn đầu là ***phép loại nhanh*** [quick rejection]: các đoạn thẳng không thể giao nhau nếu các hộp định cận của chúng không giao nhau. ***Hộp định cận*** [bounding box] của một hình hình học là hình chữ nhật nhỏ nhất chứa hình và các đoạn thẳng của nó là song song với trục x và trục y. Hộp định cận của đoạn thẳng $\overline{p_1 p_2}$ được biểu diễn bởi hình chữ nhật $(\hat{p}_1, \hat{p}_2)$ có điểm dưới trái $\hat{p}_1 = (\hat{x}_1, \hat{y}_1)$ và điểm trên phải $\hat{p}_2 = (\hat{x}_2, \hat{y}_2)$, ở đó $\hat{x}_1 = \min(x_1, x_2)$, $\hat{y}_1 = \min(y_1, y_2)$, $\hat{x}_2 =$

$\max(x_1, x_2)$, và $\hat{y}_2 = \max(y_1, y_2)$. Hai hình chữ nhật, được biểu diễn bởi các điểm dưới trái và trên phải $(\hat{p}_1, \hat{p}_2)$ và $(\hat{p}_3, \hat{p}_4)$, giao nhau nếu và chỉ nếu phép hội

$$(\hat{x}_2 \geq \hat{x}_3) \wedge (\hat{x}_4 \geq \hat{x}_1) \wedge (\hat{y}_2 \geq \hat{y}_3) \wedge (\hat{y}_4 \geq \hat{y}_1)$$

là đúng. Các hình chữ nhật phải giao nhau theo cả hai chiều. Hai phép so sánh đầu tiên trên đây xác định các hình chữ nhật có giao nhau trong x hay không; hai phép so sánh thứ hai xác định các hình chữ nhật có giao nhau trong y hay không.

Giai đoạn thứ hai để xác định hai đoạn thẳng có giao nhau hay không sẽ quyết định mỗi đoạn thẳng có "cưỡi" đường thẳng chứa đoạn thẳng kia hay không. Một đoạn thẳng $\overline{p_1p_2}$ ***cưỡi*** một đường thẳng nếu điểm p_1 nằm trên một cạnh của đường thẳng và điểm p_2 nằm trên cạnh kia. Nếu p_1 hoặc p_2 nằm trên đường thẳng, thì ta nói rằng đoạn thẳng cưỡi đường thẳng. Hai đoạn thẳng giao nhau nếu và chỉ nếu chúng qua được đợt kiểm tra phép loại nhanh và mỗi đoạn thẳng cưỡi đường thẳng chứa đoạn thẳng kia.



Hình 35.3 Xác định đoạn thẳng $\overline{p_3p_4}$ cưỡi đường thẳng chứa đoạn thẳng $\overline{p_1p_2}$. **(a)** Nếu nó cưỡi, thì các dấu của các tích vectơ $(p_3 - p_1) \times (p_2 - p_1)$ và $(p_4 - p_1) \times (p_2 - p_1)$ khác nhau. **(b)** Nếu nó không cưỡi, thì các dấu của các tích vectơ giống nhau. **(c)-(d)** Các trường hợp cận ở đó ít nhất một trong các tích vectơ là zero và đoạn thẳng cưỡi. **(e)** Một trường hợp cận ở đó các đoạn thẳng là cộng tuyến nhưng không giao nhau. Cả hai tích vectơ là zero, nhưng chúng sẽ không được thuật toán chúng ta tính toán bởi các đoạn thẳng không thể tiến hành để kiểm tra phép loại nhanh—các hộp định cận của chúng không giao nhau.

Ta có thể dùng phương pháp tích vectơ để xác định đoạn thẳng p_3p_4 có cưỡi đường thẳng chứa các điểm p_1 và p_2 hay không. Như đã nêu trong các Hình 35.3(a) và (b), ý tưởng đó là xác định các đoạn thẳng có hướng $\overrightarrow{p_1p_3}$ và $\overrightarrow{p_1p_4}$ có các hướng trái ngược tương đối với $\overrightarrow{p_1p_2}$ hay không. Nếu có, thì đoạn thẳng cưỡi đường thẳng. Nhớ lại ta có thể xác định các hướng tương đối bằng các tích vectơ, ta chỉ việc kiểm tra các dấu của các tích vectơ $(p_3 - p_1) \times (p_2 - p_1)$ và $(p_4 - p_1) \times (p_2 - p_1)$ có khác nhau hay không. Một điều kiện cận xảy ra nếu một trong hai tích vectơ là zero. Trong trường hợp này, p_3 hoặc p_4 nằm trên đường thẳng chứa đoạn thẳng $\overline{p_1p_2}$. Bởi hai đoạn thẳng đã qua đợt kiểm tra phép loại nhanh, nên trên thực tế một trong các điểm p_3 và p_4 phải nằm trên đoạn thẳng $\overline{p_1p_2}$. Hai tình huống như vậy được nêu trong các Hình 35.3(c) và (d). Trường hợp ở đó hai đoạn thẳng là cộng tuyến nhưng không giao nhau, nêu trong Hình 35.3(e), được loại bỏ bởi đợt kiểm tra phép loại nhanh. Một điều kiện cận chung cuộc xảy ra nếu một hoặc cả hai đoạn thẳng có chiều dài zero, nghĩa là, nếu các điểm cuối của nó trùng khớp. Nếu cả hai đoạn thẳng có chiều dài zero, thì chỉ cần đợt kiểm tra phép loại nhanh là đủ. Nếu chỉ một đoạn thẳng, giả sử $\overline{p_3p_4}$, có chiều dài zero, thì các đoạn thẳng giao nhau nếu và chỉ nếu tích vectơ $(p_3 - p_1) \times (p_2 - p_1)$ là zero.

Các ứng dụng khác của các tích vectơ

Các đoạn sau của chương này sẽ giới thiệu các công dụng bổ sung của các tích vectơ. Trong Đoạn 35.3, ta cần sắp xếp một tập hợp các điểm theo các góc cực của chúng đối với một gốc đã cho. Như Bài tập 35.1-2 yêu cầu bạn nêu, các tích vectơ có thể được dùng để thực hiện các phép so sánh trong thủ tục sắp xếp. Trong Đoạn 35.2, ta sẽ dùng các cây đỏ đen để duy trì cách sắp xếp dọc của một tập hợp các đoạn thẳng không giao nhau. Thay vì duy trì các giá trị khóa tường minh, ta sẽ thay mỗi so sánh khóa trong mã cây đỏ đen bằng một phép tính tích vectơ để xác định trong hai đoạn thẳng cắt một đường thẳng dọc đã cho đoạn thẳng nào nằm bên trên đoạn thẳng kia.

Bài tập

35.1-1

Chứng minh nếu $p_1 \times p_2$ là dương, thì vectơ p_1 theo chiều kim đồng hồ từ vectơ p_2 đối với gốc (0, 0) và nếu tích vectơ này là âm, thì p_1 ngược chiều kim đồng hồ từ p_2.

35.1-2

Viết mã giả để sắp xếp một dãy $\langle p_1, p_2, \ldots, p_n \rangle$ n điểm theo các góc

cực của chúng đối với một gốc điểm p_0 đã cho. Thủ tục của bạn phải mất $O(n \lg n)$ thời gian và dùng các tích vectơ để so sánh các góc.

35.1-3

Nêu cách xác định trong $O(n^2 \lg n)$ thời gian ba điểm bất kỳ trong một tập hợp n điểm có phải là cộng tuyến không.

35.1-4

Giáo sư Amundsen đề xuất phương pháp dưới đây để xác định một dãy $\langle p_0, p_1, ..., p_{n-1} \rangle$ n điểm hình thành các đỉnh liên tục của một đa giác lồi. (Xem Đoạn 16.4 để có các định nghĩa liên quan đến các đa giác.) Kết xuất "yes" nếu tập hợp $\{\angle p_i p_{i+1} p_{i+2} : i = 0, 1, ..., n-1\}$, ở đó phép cộng con chữ dưới được thực hiện modulo n, không chứa cả các lần quay trái lẫn quay phải; bằng không, kết xuất "no." Chứng tỏ mặc dù phương pháp này chạy trong thời gian tuyến tính, nó không luôn tạo ra đáp án đúng. Sửa đổi phương pháp của giáo sư sao cho nó luôn tạo ra đáp án đúng trong thời gian tuyến tính.

35.1-5

Cho một điểm $p_0 = (x_0, y_0)$, ***tia ngang phải*** từ p_0 là tập hợp các điểm $\{p_i = (x_i, y_i) : x_i \geq x_0 \text{ và } y_i = y_0\}$, nghĩa là, nó là tập hợp các điểm theo đúng hướng phải của p_0 cùng với chính p_0. Nêu cách xác định một tia ngang phải đã cho từ po có cắt một đoạn thẳng $\overline{p_1 p_2}$ trong $O(1)$ thời gian hay không bằng cách rút gọn bài toán thành bài toán xác định hai đoạn thẳng có giao nhau không.

35.1-6

Một cách để xác định một điểm p_0 có nằm trong phần trong của một đa giác P đơn nhưng không nhất thiết là lồi hay không đó là xem xét bất kỳ tia nào từ p_0 và kiểm tra tia đó cắt biên của P theo một số lần lẻ nhưng chính p_0 không nằm trên biên của P. Nêu cách tính toán trong $\Theta(n)$ thời gian một điểm p_0 có nằm trong phần trong của một đa giác n-đỉnh P hay không. (*Mách nước:* Dùng Bài tập 35.1-5. Bảo đảm thuật toán là đúng khi tia cắt biên đa giác tại một đỉnh và khi tia phủ chồng một cạnh của đa giác-)

35.1-7

Nêu cách tính toán diện tích của một đa giác đơn n-đỉnh, nhưng không nhất thiết là lồi, trong $\Theta(n)$ thời gian.

35.2 Xác định một cặp bất kỳ có giao nhau không

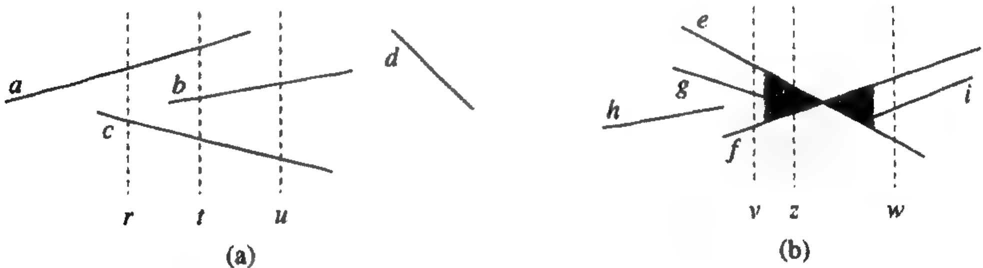
Đoạn này trình bày một thuật toán để xác định hai đoạn thẳng bất kỳ trong một tập hợp các đoạn thẳng có giao nhau hay không. Thuật toán

sử dụng một kỹ thuật có tên là "phép quét," là chung cho nhiều thuật toán hình học điện toán. Hơn nữa, như các bài tập ở cuối đoạn này cho thấy, thuật toán này, hoặc các biến thể đơn giản của nó, có thể được dùng để giải các bài toán hình học điện toán khác.

Thuật toán chạy trong $O(n \lg n)$ thời gian, ở đó n là số lượng đoạn thẳng mà ta có. Nó chỉ xác định có điểm giao nào tồn tại hay không; nó không in tất cả các giao điểm. (Theo Bài tập 35.2-1, nó chiếm $\Omega(n^2)$ thời gian trong trường hợp xấu nhất để tìm *tất cả* các giao điểm trong một tập hợp n đoạn thẳng.)

Trong ***phép quét***, một ***đường quét*** dọc ảo đi qua một tập hợp các đối tượng hình học đã cho, thường từ trái qua phải. Chiều không gian mà đường quét dời qua, trong trường hợp này là chiều x, được xem là một chiều của thời gian. Phép quét cung cấp một phương pháp để sắp xếp các đối tượng hình học, thường là bằng cách đặt chúng vào một cấu trúc dữ liệu động, và để vận dụng các mối quan hệ giữa chúng. Thuật toán giao đoạn thẳng trong đoạn này xét tất cả các điểm cuối đoạn thẳng theo thứ tự từ trái qua phải và kiểm tra một điểm giao mỗi lần nó gặp một điểm cuối.

Thuật toán của chúng ta để xác định hai bất kỳ trong số n đoạn thẳng có giao nhau hay không đã thực hiện hai giả thiết đơn giản hóa. Thứ nhất, ta mặc nhận rằng không đoạn thẳng đầu vào nào là dọc. Thứ hai, ta mặc nhận rằng không có ba đoạn thẳng đầu vào nào giao nhau tại một điểm đơn lẻ. (Bài tập 35.2-8 yêu cầu bạn mô tả một kiểu thực thi làm việc cho dù các giả thiết này không đứng vững.) Quả thực, việc gỡ bỏ các giả thiết đơn giản hóa như vậy và việc đối phó với các điều kiện cận thường là phần khó nhất của lập trình các thuật toán hình học điện toán và chứng minh tính đúng đắn của chúng.



Hình 35.4 Cách sắp xếp giữa các đoạn thẳng tại các đường quét dọc khác nhau. **(a)** Ta có $a >_r c$, $a >_t b$, $b >_t c$, $a >_t c$, và $b >_u c$. Đoạn thẳng d không so sánh được với các đoạn thẳng khác đã nêu. **(b)** Khi các đoạn thẳng a và f giao nhau, các thứ tự của chúng được đảo ngược: ta có $e >_v f$ nhưng $f >_w e$. Mọi đường quét (như z) đi qua vùng tô bóng đều có e và f liên tiếp trong thứ tự tổng của nó.

Sắp xếp các đoạn thẳng

Bởi ta mặc nhận không có các đoạn thẳng dọc, nên mọi đoạn thẳng đầu vào cắt một đường quét dọc đã cho sẽ cắt nó tại một điểm đơn lẻ. Như vậy, ta có thể sắp xếp các đoạn thẳng cắt một đường quét dọc theo các tọa độ y của các điểm giao.

Để chính xác hơn, ta xét hai đoạn thẳng không giao nhau s_1 và s_2. Ta nói rằng các đoạn thẳng này là ***so sánh được*** tại x nếu đường quét dọc có x tọa độ-x cắt cả hai trong số chúng. Ta nói rằng s_1 nằm ***bên trên*** s_2 tại x, được viết $s_1 >_x s_2$, nếu s_1 và s_2 so sánh được tại x và điểm giao của s_1 với đường quét tại x cao hơn điểm giao của s_2 với cùng đường quét. Trong Hình 35.4(a), ví dụ, ta có các mối quan hệ $a >_r c$, $a >_t b$, $b >_t c$, $a >_t c$, và $b >_u c$. Đoạn thẳng d không so sánh được với bất kỳ đoạn thẳng nào khác.

Với bất kỳ x đã cho, hệ thức "$>_x$" là một thứ tự tổng (xem Đoạn 5.2) trên các đoạn thẳng cắt đường quét tại x. Tuy nhiên, thứ tự có thể khác nhau với các giá trị khác nhau của x, khi các đoạn thẳng nhập và rời cách sắp xếp thứ tự. Một đoạn thẳng nhập cách sắp xếp thứ tự khi đường quét gặp điểm cuối trái của nó, và nó rời cách sắp xếp thứ tự khi gặp điểm cuối phải của nó.

Điều gì xảy ra khi đường quét đi qua điểm giao của hai đoạn thẳng? Như Hình 35.4(b) đã nêu, các vị trí của chúng trong thứ tự tổng được đảo ngược. Các đường quét v và w nằm về bên trái và phải, theo thứ tự nêu trên, của điểm giao của các đoạn thẳng e và f, và ta có $e >_v f$ và $f >_w e$. Lưu ý, bởi ta mặc nhận rằng không có ba đoạn thẳng giao nhau tại cùng một điểm, nên phải có một đường quét dọc x mà các đoạn thẳng giao nhau e và f là ***liên tiếp*** theo thứ tự tổng $>_x$. Mọi đường quét đi qua vùng tô bóng của Hình 35.4(b), như z, đều có e và f liên tiếp trong thứ tự tổng của nó.

Dời đường quét

Các thuật toán quét thường quản lý hai tập hợp dữ liệu:

1. ***Tình trạng đường quét*** cho các mối quan hệ giữa các đối tượng giao nhau bởi đường quét.

2. ***Lịch biểu điểm sự kiện*** là một dãy các tọa độ x, được sắp xếp từ trái qua phải, định nghĩa các vị trí dừng của đường quét. Ta gọi mỗi vị trí dừng như vậy là một ***điểm sự kiện***. Các thay đổi đối với tình trạng đường quét chỉ xảy ra tại các điểm sự kiện.

Với vài thuật toán (ví dụ, thuật toán được yêu cầu trong Bài tập 35.2-7), lịch biểu điểm sự kiện được xác định động khi thuật toán tiến triển.

Tuy nhiên, thuật toán ở đây xác định các điểm sự kiện một cách tĩnh, chỉ dựa trên các tính chất đơn giản của đầu vào. Nói cụ thể, mỗi điểm cuối của đoạn thẳng là một điểm sự kiện. Ta sắp xếp các điểm cuối đoạn thẳng bằng cách tăng tọa độ x và tiến hành từ trái qua phải. Ta chèn một đoạn thẳng vào tình trạng đường quét khi gặp điểm cuối trái của nó, và xóa nó ra khỏi tình trạng đường quét khi gặp điểm cuối của nó. Mỗi khi hai đoạn thẳng trở thành liên tiếp lần đầu tiên trong thứ tự tổng, ta kiểm tra xem chúng có giao nhau không.

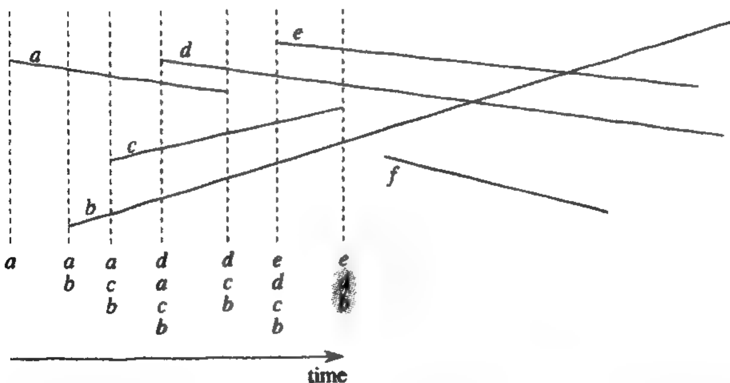
Tình trạng đường quét là một thứ tự tổng T, mà ta yêu cầu các phép toán dưới đây:

- INSERT(T, s): chèn đoạn thẳng s vàoT.
- DELETE(T, s): xóa đoạn thẳng s ra khỏi T.
- ABOVE(T, s): trả về đoạn thẳng ngay bên trên đoạn thẳng s trong T.
- BELOW (T, s): trả về đoạn thẳng ngay bên dưới đoạn thẳng s trong T.

Nếu có n đoạn thẳng trong đầu vào, ta có thể thực hiện từng trong số các phép toán trên đây trong $O(\lg n)$ thời gian dùng các cây đỏ đen. Hãy nhớ lại các phép toán cây đỏ đen trong Chương 14 liên quan đến việc so sánh các khóa. Ta có thể thay các phép so sánh khóa bằng các phép so sánh tích vectơ xác định cách sắp xếp tương đối của hai đoạn thẳng (xem Bài tập 35.2-2).

Mã giả điểm giao đoạn thẳng

Thuật toán dưới đây nhận một tập hợp S n đoạn thẳng làm đầu vào.



Hình 35.5 Sự thi hành của ANY-SEGMENTS-INTERSECT. Mỗi đường chấm cách là đường quét tại một điểm sự kiện, và cách sắp xếp thứ tự của các tên đoạn thẳng bên dưới mỗi đường quét là thứ tự tổng T tại cuối vòng lặp **for** ở đó điểm sự kiện tương ứng được xử lý. Điểm giao của các đoạn thẳng d và b được tìm thấy khi xóa đoạn thẳng c.

trả về giá trị Bool TRUE nếu mọi cặp đoạn thẳng trong S giao nhau, và bằng không là FALSE. Thứ tự tổng T được thực thi bởi một cây đỏ đen.

```
ANY-SEGMENTS-INTERSECT(S)
1   T ← ∅
2   sắp xếp các điểm cuối của các đoạn thẳng trong S từ trái qua phải,
            ngắt các mối ràng buộc bằng cách đặt các điểm có các
                         tọa độ y thấp hơn trước
3   for mỗi điểm p trong danh sách đã sắp xếp của các điểm cuối
4        do if p là điểm cuối trái của một đoạn thẳng s
5              then INSERT(T, s)
6                   if (ABOVE(T, s) tồn tại và cắt s)
                        hoặc (BELOW(T,s) tồn tại và cắt s)
7                     then return TRUE
8           if p là điểm cuối phải của một đoạn thẳng s
9              then if cả ABOVE(T,s) lẫn BELOW(T,s) tồn tại
                      và ABOVE(T, s) cắt BELOW(T, s)
10                    then return TRUE
11                 DELETE(T, s)
12  return FALSE
```

Hình 35.5 minh họa việc thi hành thuật toán. Dòng 1 khởi tạo thứ tự tổng là trống. Dòng 2 xác định lịch biểu điểm sự kiện bằng cách sắp xếp $2n$ điểm cuối đoạn thẳng từ trái qua phải, ngắt các mối ràng buộc bằng cách đặt các điểm có các tọa độ y thấp hơn trước. Lưu ý, dòng 2 có thể được thực hiện bằng cách sắp xếp các điểm cuối theo thứ tự từ điển trên (x, y).

Mỗi lần lặp lại của vòng lặp **for** của các dòng 3-11 sẽ xử lý một điểm sự kiện p. Nếu p là điểm cuối trái của một đoạn thẳng s, dòng 5 bổ sung s vào thứ tự tổng, và các dòng 6-7 trả về TRUE nếu s cắt một trong hai đoạn thẳng mà nó liền kề theo thứ tự tổng được định nghĩa bằng đường quét đi qua p.

(Một điều kiện cận xảy ra nếu p nằm trên một đoạn thẳng s' khác. Trong trường hợp này, ta chỉ yêu cầu s và s' được đặt liền kề vào T.) Nếu p là điểm cuối phải của một đoạn thẳng s, thì s phải được được xóa ra khỏi thứ tự tổng. Các dòng 9-10 trả về TRUE nếu có một điểm giao giữa các đoạn thẳng bao quanh s trong thứ tự tổng được định nghĩa bởi đường quét đi qua p; các đoạn thẳng này sẽ trở thành liền kề trong thứ

tự tổng khi s được xóa. Nếu các đoạn thẳng này không giao nhau, dòng 11 xóa đoạn thẳng s ra khỏi thứ tự tổng. Cuối cùng, nếu không tìm thấy các giao điểm trong khi xử lý tất cả $2n$ điểm sự kiện, dòng 12 trả về FALSE.

Tính đúng đắn

Định lý dưới đây chứng tỏ ANY-SEGMENTS-INTERSECT là đúng.

Định lý 35.1

Lệnh gọi ANY-SEGMENTS-INTERSECT(S) trả về TRUE nếu và chỉ nếu có một điểm giao giữa các đoạn thẳng trong S.

Chứng minh Thủ tục chỉ có thể không đúng bằng cách trả về TRUE khi không có điểm giao nào tồn tại hoặc trả về FALSE khi có ít nhất một điểm giao. Trường hợp trước không thể xảy ra, bởi ANY-SEGMENTS-INTERSECT chỉ trả về TRUE nếu nó tìm thấy một điểm giao giữa hai trong số các đoạn thẳng đầu vào.

Để chứng tỏ trường hợp sau không thể xảy ra, vì sự mâu thuẫn ta hãy giả sử có ít nhất một điểm giao, và hơn nữa ANY-SEGMENTS-INTERSECT trả về FALSE. Cho p là điểm giao mút trái, ngắt các mối ràng buộc bằng cách chọn cái có tọa độ y thấp nhất, và cho a và b là các đoạn thẳng giao nhau tại p. Bởi không có các giao điểm xảy ra về bên trái của p, nên thứ tự mà T cung cấp là đúng tại tất cả các điểm về bên trái của p. Bởi không có ba đoạn thẳng giao nhau tại cùng một điểm, ở đó tồn tại một đường quét z nơi a và b trở thành liền kề trong thứ tự tổng.[3] Hơn nữa, z nằm về bên trái của p hoặc đi qua p. Ở đó tồn tại một điểm cuối đoạn thẳng q trên đường quét z là điểm sự kiện nơi a và b trở thành liền kề trong thứ tự tổng. Nếu p nằm trên đường quét z, thì $q = p$. Nếu p không nằm trên đường quét z, thì q nằm bên trái của p. Trong cả hai trường hợp, thứ tự mà T cung cấp là đúng ngay trước khi q được xử lý. (Ở đây ta dựa vào p là thấp nhất trong số các điểm giao mút trái. Bởi thứ tự từ điển ở đó các điểm sự kiện được xử lý, nên cho dù p nằm trên đường quét z và có một điểm giao p' khác trên z, điểm sự kiện $q = p$ được xử lý trước khi điểm giao p' kia có thể gây trở ngại cho thứ tự tổng T.) Chỉ có hai khả năng để hành động diễn ra tại điểm sự kiện q:

1. Hoặc a hoặc b được chèn vào T, và đoạn thẳng kia nằm bên trên hoặc bên dưới nó trong thứ tự tổng. Các dòng 4-7 phát hiện trường hợp này.

---

[3] Nếu ta cho phép ba đoạn thẳng giao nhau tại cùng một điểm, có thể có một đoạn thẳng c xen vào cắt cả a và b tại điểm p. Nghĩa là, ta có thể có một $a <_w c$ và $c <_w b$ với tất cả các đường quét w về bên trái của p mà $a <_w b$.

2. Các đoạn thẳng *a* và *b* nằm sẵn trong *T*, và một đoạn thẳng giữa chúng trong thứ tự tổng được xóa, khiến *a* và *b* trở thành liền kề. Các dòng 8-11 phát hiện trường hợp này.

Với một trong trường hợp, điểm giao *p* được tìm thấy, mâu thuẫn với giả thiết rằng thủ tục trả về FALSE.

Thời gian thực hiện

Nếu có *n* đoạn thẳng trong tập hợp *S*, thì ANY-SEGMENTS-INTERSECT chạy trong thời gian $O(n \lg n)$. Dòng 1 chiếm $O(1)$ thời gian. Dòng 2 chiếm $O(n \lg n)$ thời gian, dùng kỹ thuật sắp xếp trộn hoặc sắp xếp đống. Bởi có $2n$ điểm sự kiện, vòng lặp **for** của các dòng 3-11 lặp lại tối đa $2n$ lần. Mỗi lần lặp lại mất $O(\lg n)$ thời gian, bởi mỗi phép toán cây đỏ đen mất $O(\lg n)$ thời gian và, dùng phương pháp của Đoạn 35.1, mỗi đợt kiểm tra điểm giao mất $O(1)$ thời gian. Như vậy, tổng thời gian là $O(n \lg n)$.

Bài tập

35.2-1

Chứng tỏ có thể có $\Theta(n^2)$ giao điểm trong một tập hợp *n* đoạn thẳng.

35.2-2

Cho hai đoạn thẳng không giao nhau *a* và *b* so sánh được tại *x*, hãy nêu cách sử dụng các tích vectơ để xác định trong $O(1)$ thời gian cái nào trong $a >_x b$ hoặc $b >_x a$ đứng vững.

35.2-3

Giáo sư Maginot đề xuất sửa đổi ANY-SEGMENTS-INTERSECT sao cho thay vì trở về vào lúc tìm một điểm giao, nó in các đoạn thẳng giao nhau và tiếp tục đến lần lặp lại kế tiếp của vòng lặp **for**. Giáo sư gọi thủ tục kết quả là PRINT-INTERSECTING-SEGMENTS và cho rằng nó in tất cả các giao điểm, từ trái qua phải, như chúng xảy ra trong tập hợp các đoạn thẳng. Chứng tỏ giáo sư đã sai trên hai điểm bằng cách cho một tập hợp các đoạn thẳng mà điểm giao đầu tiên được tìm thấy bởi PRINT-INTERSECTING-SEGMENTS không phải là điểm giao mút trái và một tập hợp các đoạn thẳng mà PRINT-INTERSECTING-SEGMENTS không tìm ra tất cả các giao điểm.

35.2-4

Nêu một thuật toán $O(n \lg n)$ thời gian để xác định một đa giác n-đỉnh có phải là đơn hay không.

35.2-5

Nêu một thuật toán $O(n \lg n)$ thời gian để xác định hai đa giác đơn có một tổng n đỉnh có giao nhau hay không.

35.2-6

Một ***đĩa*** bao gồm một vòng tròn cộng với phần trong của nó và được biểu thị bằng tâm điểm và bán kính của nó. Hai đĩa giao nhau nếu chúng có bất kỳ điểm chung nào. Nêu một thuật toán $O(n \lg n)$ thời gian để xác định hai đĩa bất kỳ trong một tập hợp n có giao nhau hay không.

35.2-7

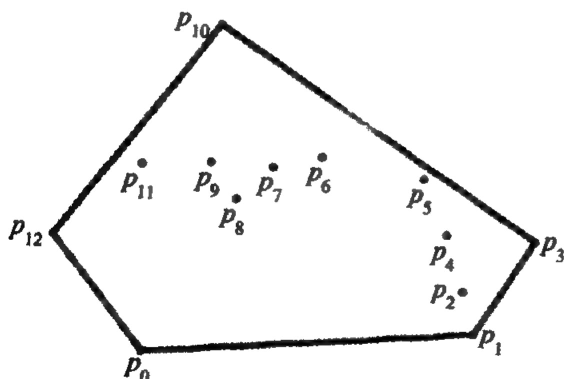
Cho một tập hợp n đoạn thẳng chứa một tổng k giao điểm, hãy nêu cách kết xuất tất cả k giao điểm trong $O((n + k) \lg n)$ thời gian.

35.2-8

Nêu cách thực thi các thủ tục cây đỏ đen sao cho ANY-SEGMENTS-INTERSECT làm việc đúng đắn cho dù có vài đoạn thẳng là dọc hoặc trên ba đoạn thẳng giao nhau tại cùng một điểm. Chứng minh kiểu thực thi của bạn là đúng.

35.3 Tìm bao lồi

Bao lồi của một tập hợp Q gồm các điểm là đa giác lồi nhỏ nhất P



Hình 35.6 Một tập hợp các điểm Q có bao lồi của nó CH(Q) ở dạng màu xám.

mà mỗi điểm trong Q nằm trên biên của P hoặc trong phần trong của nó. Ta thể hiện bao lồi của Q bằng CH(Q). Theo trực giác, ta có thể xem mỗi điểm trong Q như là một chiếc đinh lòi ra từ một tấm ván. Như vậy bao lồi là hình dáng được lập thành bởi một dải cao su chặt bao quanh tất cả các chiếc đinh. Hình 35.6 có nêu một tập hợp các điểm và bao lồi của nó.

Trong đoạn này, ta sẽ trình bày hai thuật toán tính toán bao lồi của một tập hợp n điểm. Cả hai thuật toán kết xuất các đỉnh của bao lồi trong thứ tự ngược chiều kim đồng hồ. Thuật toán đầu tiên, có tên là quét Graham [Graham's scan], chạy trong $O(n \lg n)$ thời gian. Thuật toán thứ hai, có tên diễu hành Jarvis [Jarvis's march], chạy trong $O(nh)$ thời gian, ở đó h là số lượng các đỉnh của bao lồi. Như đã thấy trong Hình 35.6, mọi đỉnh của CH(Q) là một điểm trong Q. Cả hai thuật toán đều khai thác tính chất này, quyết định các đỉnh nào trong Q được giữ lại dưới dạng các đỉnh của bao lồi và các đỉnh nào trong Q được vứt bỏ.

Thực tế, có vài phương pháp tính toán các bao lồi trong $O(n \lg n)$ thời gian. Cả thuật toán quét Graham lẫn diễu hành Jarvis đều dùng một kỹ thuật tên "quét quay" [rotational sweep], xử lý các đỉnh theo thứ tự của các góc cực mà chúng hình thành với một đỉnh tham chiếu. Các phương pháp khác bao gồm:

- Trong ***phương pháp gia số,*** các điểm được sắp xếp từ trái qua phải, cho ra một dãy $\langle p_1, p_2, ..., p_n \rangle$. Tại giai đoạn thứ i, bao lồi CH($\{p_1, p_2, ..., p_{i-1}\}$) gồm $i - 1$ điểm mút trái được cập nhật theo điểm thứ i từ bên trái, như vậy hình thành CH($\{p_1, p_2, ..., p_i\}$). Như Bài tập 35.3-6 yêu cầu bạn nêu, phương pháp này có thể được thực thi để lấy một tổng $O(n \lg n)$ thời gian.

- Trong ***phương pháp chia để trị,*** trong thời gian $\Theta(n)$, tập hợp n điểm được chia thành hai tập hợp con, một gồm $\lceil n/2 \rceil$ điểm mút trái và một gồm $\lfloor n/2 \rfloor$ điểm mút phải, các bao lồi của các tập hợp con được tính toán một cách đệ quy, rồi một phương pháp thông minh hơn được dùng để tổ hợp các bao trong $O(n)$ thời gian.

- ***Phương pháp lược tìm*** [prune-and-search] tương tự như thuật toán trung tuyến thời gian tuyến tính trường hợp xấu nhất trong Đoạn 10.3. Nó tìm phần trên (hoặc "xích trên") của bao lồi bằng cách liên tục vứt

bỏ một phân số bất biến của các điểm còn lại cho đến khi chỉ còn lại xích trên [upper chain] của bao lồi. Sau đó, nó cũng thực hiện như vậy với xích dưới. Phương pháp này theo tiệm cận là nhanh nhất: nếu bao lồi chứa h đỉnh, nó chỉ chạy trong $O(n \lg h)$ thời gian.

Việc tính toán bao lồi của một tập hợp các điểm là một bài toán đáng quan tâm với tư cách cá nhân của nó. Hơn nữa, các thuật toán cho một số bài toán hình học điện toán khác bắt đầu bằng cách tính toán một bao lồi. Ví dụ, xét ***bài toán cặp xa nhất*** hai chiều: ta có một tập hợp n điểm trong mặt phẳng và muốn tìm hai điểm mà khoảng cách giữa chúng là cực đại. Như Bài tập 35.3-3 yêu cầu bạn chứng minh, hai điểm này phải là các đỉnh của bao lồi. Mặc dù ta không chứng minh nó ở đây, song cặp đỉnh xa nhất của một đa giác lồi n-đỉnh có thể tìm thấy trong $O(n)$ thời gian. Như vậy, bằng cách tính toán bao lồi của n điểm đầu vào trong $O(n \lg n)$ thời gian rồi tìm cặp xa nhất của các đỉnh đa giác lồi kết quả, ta có thể tìm ra cặp điểm xa nhất trong một tập hợp bất kỳ n điểm trong $O(n \lg n)$ thời gian.

Quét Graham

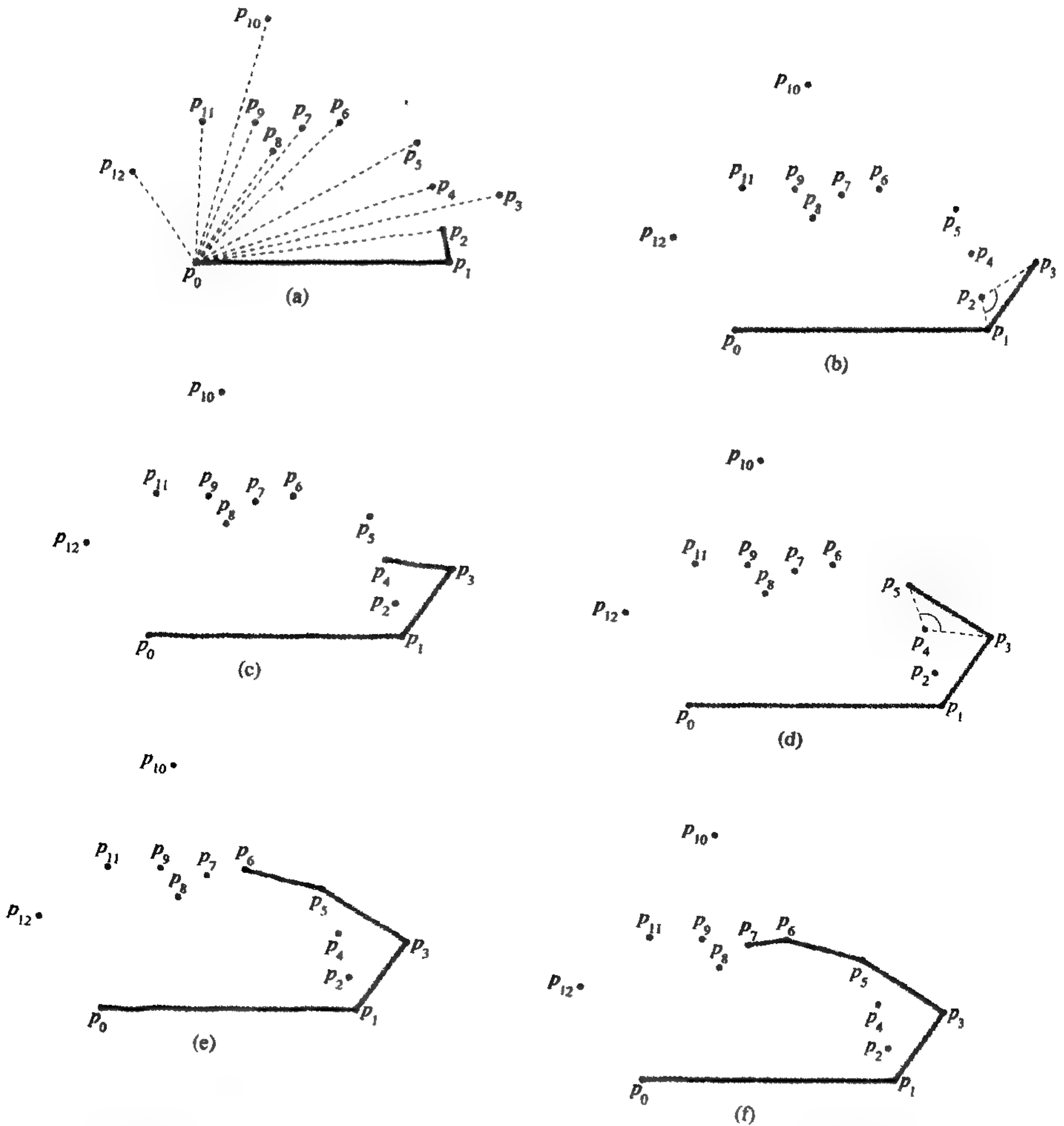
Quét Graham giải bài toán bao lồi bằng cách duy trì một ngăn xếp S các điểm ứng viên. Mỗi điểm của tập hợp đầu vào Q được đẩy một lần lên trên ngăn xếp, và các điểm không phải là các đỉnh của CH(Q) chung cuộc được kéo ra từ ngăn xếp. Khi thuật toán kết thúc, ngăn xếp S chứa chính xác các đỉnh của CH(Q), theo thứ tự ngược chiều kim đồng hồ mà chúng xuất hiện trên biên.

Thủ tục GRAHAM-SCAN nhận một tập hợp Q các điểm làm đầu vào, ở đó $|Q| \geq 3$. Nó gọi các hàm TOP(S), trả về điểm trên đầu ngăn xếp S mà không thay đổi S, và NEXT-TO-TOP(S), trả về điểm một khoản nhập bên dưới đỉnh ngăn xếp S mà không thay đổi S. Như ta sẽ chứng minh dưới đây, ngăn xếp S mà GRAHAM-SCAN trả về sẽ chứa, từ dưới lên, chính xác các đỉnh của CH(Q) theo thứ tự ngược chiều kim đồng hồ.

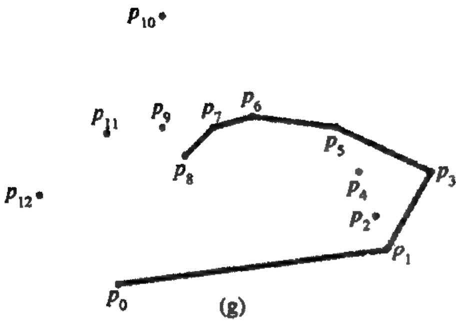
GRAHAM-SCAN (Q)

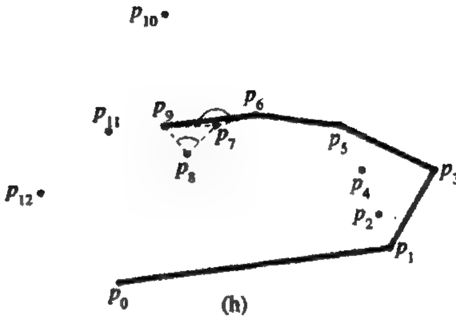
1 cho p_0 là điểm trong Q có tọa độ y cực tiểu,

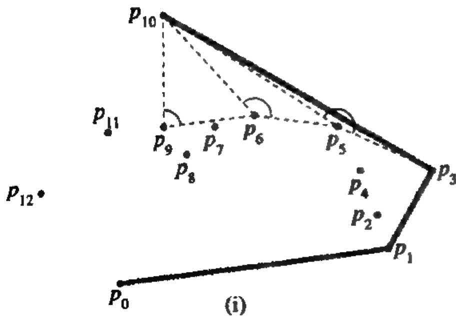
hoặc điểm mút trái như vậy trong trường hợp của một ràng buộc

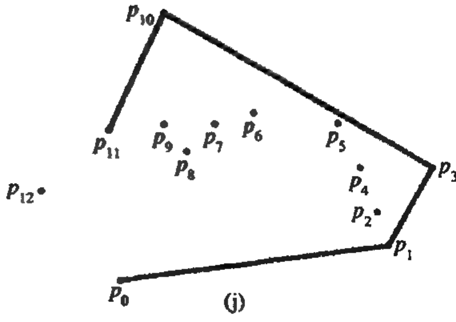


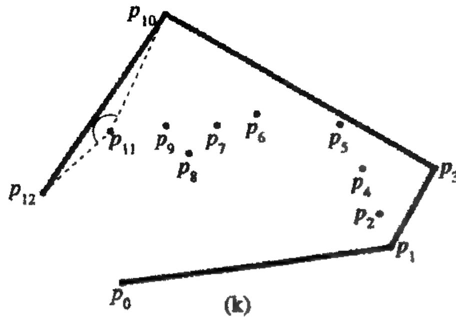
Hình 35.7 Tiến trình thi hành của GRAHAM-SCAN trên tập hợp Q của Hình 35.6. Bao lồi hiện hành chứa trong ngăn xếp S được nêu theo màu xám tại mỗi bước. **(a)** Các góc cực có sắp xếp của $\langle p_1, p_2, \ldots, p_{12} \rangle$ tương đối với p_0 và ngăn xếp ban đầu S chứa p_0, p_1, và p_2. **(b)-(k)** Ngăn xếp S sau mỗi lần lặp lại của vòng lặp **for** của các dòng 7-10. Các đường chấm cách nêu các lần quay phi trái [nonleft turns], khiến các điểm được kéo ra từ ngăn xếp. Ví dụ, trong phần (h), lần quay phải tại góc $\angle p_7 p_8 p_9$ khiến p_8 được kéo ra, rồi lần quay phải tại góc $\angle p_6 p_7 p_9$ khiến p_7 được kéo ra. **(l)** Bao lồi được trả về bởi thủ tục, so khớp với trong Hình 35.6.

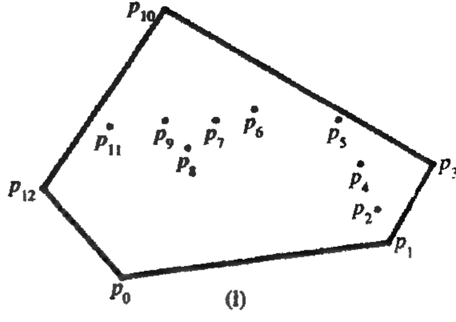

p_{10}
p_{11}
p_9
p_7
p_6
p_8
p_5
p_3
p_{12}
p_4
p_2
p_1
p_0
(g)


p_{10}
p_{11}
p_9
p_6
p_7
p_8
p_5
p_3
p_{12}
p_4
p_2
p_1
p_0
(h)


p_{10}
p_{11}
p_9
p_7
p_6
p_5
p_8
p_3
p_{12}
p_4
p_2
p_1
p_0
(i)


p_{10}
p_{11}
p_9
p_7
p_6
p_5
p_8
p_3
p_{12}
p_4
p_2
p_1
p_0
(j)


p_{10}
p_{11}
p_9
p_7
p_6
p_5
p_8
p_3
p_{12}
p_4
p_2
p_1
p_0
(k)


p_{10}
p_{11}
p_9
p_7
p_6
p_5
p_8
p_3
p_{12}
p_4
p_2
p_1
p_0
(l)

2 cho $<p_1, p_2, ..., p_m>$ là các điểm còn lại trong Q,

được sắp xếp theo góc cực theo thứ tự ngược chiều kim đồng hồ quanh p_0 (nếu có nhiều điểm có cùng góc, gỡ bỏ tất cả ngoại trừ điểm xa nhất với p_0)

3 *top*[*S*] ← 0

4 PUSH(p_0, *S*)

5 PUSH(p_1, *S*)

6 PUSH(p_2, *S*)

7 **for** $i \leftarrow 3$ **to** m

8 **do while** góc được lập thành bởi các điểm NEXT-TO-TOP(*S*), TOP(*S*), và p_i thực hiện một lần quay phi trái

9 **do** POP(*S*)

10 PUSH(*S*, p_i)

11 **return** *S*

Hình 35.7 minh họa tiến độ của GRAHAM-SCAN. Dòng 1 chọn điểm p_0 làm điểm có tọa độ y thấp nhất, chọn điểm mút trái như vậy trong trường hợp của một ràng buộc. Bởi không có điểm nào trong Q nằm bên dưới p_0 và bất kỳ điểm nào khác có cùng tọa độ y nằm về bên phải của nó, nên p_0 là một đỉnh của CH(Q). Dòng 2 sắp xếp các điểm còn lại của Q theo góc cực tương đối với p_0, dùng cùng phương pháp—so sánh các tích vectơ—như trong Bài tập 35.1-2. Nếu hai hoặc nhiều điểm có cùng góc cực tương đối với p_0, tất cả ngoại trừ điểm xa nhất như vậy đều là các tổ hợp lồi của p_0 và điểm xa nhất, và do đó ta loại bỏ chúng hoàn toàn ra khỏi phần nghiên cứu. Ta cho m thể hiện số lượng các điểm khác ngoài p_0 giữ lại. Góc cực, được đo bằng radian, của mỗi điểm trong Q tương đối với p_0 nằm trong quãng nửa mở $[0, \pi/2)$. Bởi các góc cực gia tăng theo dạng ngược chiều kim đồng hồ, nên các điểm được sắp xếp theo thứ tự ngược chiều kim đồng hồ tương đối với p_0. Ta chỉ định dãy các điểm đã sắp xếp này bằng $<p_1, p_2, ..., p_m>$. Lưu ý, các điểm p_1 và p_m là các đỉnh của CH(Q) (xem Bài tập 35.3-1). Hình 35.7(a) nêu các điểm của Hình 35.6, với các góc cực có sắp xếp của $<p_1, p_2, ..., p_{12})$ tương đối với p_0.

Phần còn lại của thủ tục sử dụng ngăn xếp S. Các dòng 3-6 khởi tạo ngăn xếp để chứa, từ dưới lên, ba điểm đầu tiên p_0, p_1, và p_2. Hình 35.7(a) nêu ngăn xếp ban đầu S. Vòng lặp **for** của các dòng 7-10 lặp lại một lần cho mỗi điểm trong dãy con $\langle p_3, p_4, ..., p_m \rangle$. Chủ trương đó là sau khi xử lý điểm p_i, ngăn xếp S chứa, từ dưới lên, các đỉnh của CH($\{p_0, p_1, ..., p_i\}$) theo thứ tự ngược chiều kim đồng hồ. Vòng lặp **while** của các dòng 8-9 gỡ bỏ các điểm ra khỏi ngăn xếp nếu chúng được tìm thấy không phải là các đỉnh của bao lồi. Khi băng ngang bao lồi ngược chiều kim đồng hồ, ta phải thực hiện một lần quay trái tại mỗi đỉnh. Như vậy, mỗi lần vòng lặp **while** tìm một đỉnh nơi ta thực hiện một lần quay phi trái, đỉnh được kéo ra ra khỏi ngăn xếp. (Bằng cách kiểm tra một lần quay phi trái, thay vì chỉ là một lần quay phải, đợt kiểm tra này loại trừ khả năng của một góc bẹt tại một đỉnh của bao lồi kết quả. Đây chính là cái ta muốn, bởi mọi đỉnh của một đa giác lồi không được là một tổ hợp lồi của các đỉnh khác của đa giác.) Sau khi ta kéo ra tất cả các đỉnh có các lần quay phi trái khi tiến đến điểm p_i, ta đẩy p_i lên trên ngăn xếp. Các Hình 35.7(b)-(k) nêu trạng thái của ngăn xếp S sau mỗi lần lặp lại của vòng lặp **for**. Cuối cùng, GRAHAM-SCAN trả về ngăn xếp S trong dòng 11. Hình 35.7(l) nêu bao lồi tương ứng.

Định lý dưới đây chính thức chứng minh tính đúng đắn của GRAHAM-SCAN.

Định lý 35.2 (Tính đúng đắn của quét Graham)

Nếu GRAHAM-SCAN chạy trên một tập hợp Q các điểm, ở đó $|Q| \geq 3$, thì một điểm của Q nằm trên ngăn xếp S vào lúc kết thúc nếu và chỉ nếu nó là một đỉnh của CH(Q).

Chứng minh Như đã lưu ý trên đây, một đỉnh là một tổ hợp lồi của p_0 và một đỉnh nào khác trong Q sẽ không phải là một đỉnh của CH(Q). Một đỉnh như vậy không được gộp trong dãy $\langle p_1, p_2, ..., p_m \rangle$, và do đó nó có thể không bao giờ xuất hiện trên ngăn xếp S.

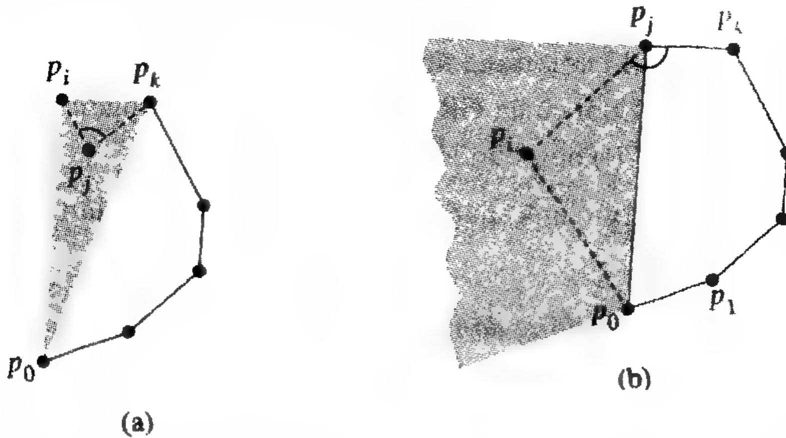
Điểm then chốt của phần chứng minh nằm trong hai tình huống nêu trong Hình 35.8. Phần (a) đối phó với các lần quay phi trái, và phần (b) đối phó với các lần quay trái.

Trước tiên, ta chứng tỏ mỗi điểm được kéo ra khỏi ngăn xếp S không phải là một đỉnh của CH(Q). Giả sử rằng điểm p_j được kéo ra khỏi ngăn

xếp bởi vì góc $\angle p_k p_j p_i$ thực hiện một lần quay phi trái, như đã nêu trong Hình 35.8(a). Bởi vì ta quét các điểm theo thứ tự của góc cực gia tăng tương đối với điểm p_0, nên có một tam giác $\triangle p_0 p_i p_k$ với điểm p_j nằm trong phần trong của tam giác hoặc trên đoạn thẳng $\overline{p_i p_k}$. Với một trong hai trường hợp, điểm p_j không thể là một đỉnh của CH(Q).

Giờ đây ta chứng tỏ mỗi điểm trên ngăn xếp S là một đỉnh của CH(Q) vào lúc kết thúc. Để bắt đầu, ta chứng minh biện luận sau đây: GRAHAM-SCAN duy trì sự bất biến rằng các điểm trên ngăn xếp S luôn hình thành các đỉnh của một đa giác lồi theo thứ tự ngược chiều kim đồng hồ.

Biện luận đứng vững ngay sau khi thi hành dòng 6, bởi các điểm p_0, p_1, và p_2 hình thành một đa giác lồi. Giờ đây ta xét cách thức mà ngăn xếp S thay đổi trong khi thi hành GRAHAM-SCAN. Các điểm được kéo ra hoặc được đẩy lên ngăn xếp. Trong trường hợp đầu, ta dựa vào một tính chất hình học đơn giản: nếu một đỉnh được gỡ bỏ ra khỏi một đa giác lồi, đa giác kết quả là lồi. Như vậy, việc kéo một điểm ra khỏi ngăn xếp S bảo toàn sự bất biến.



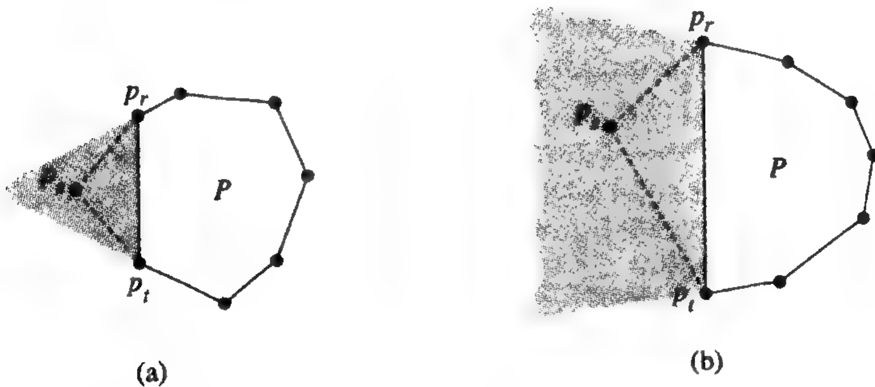
Hình 35.8 Hai tình huống cơ bản trong phần chứng minh về tính đúng đắn của GRAHAM-SCAN. **(a)** Chứng tỏ một điểm được kéo ra khỏi ngăn xếp trong GRAHAM-SCAN không phải là một đỉnh của CH(Q). Nếu điểm p_j được kéo ra khỏi ngăn xếp bởi góc $\angle p_k p_j p_i$ thực hiện một lần quay phi trái, thì tam giác tô bóng $\triangle p_0 p_k p_i$ chứa điểm p_j. Do đó, điểm p_j không phải là một đỉnh của CH(Q). **(b)** Nếu điểm p_i được đẩy lên ngăn xếp, thì đó phải là một lần quay trái tại góc $\angle p_k p_j p_i$. Bởi pi theo p_j trong cách sắp xếp góc cực của các điểm và bởi cách p_0 được chọn, nên p_i phải nằm trong vùng tô bóng. Nếu các điểm trên ngăn xếp hình thành một đa giác lồi trước khi đẩy, thì chúng phải hình thành một đa giác lồi sau đó.

Trước khi xét trường hợp ở đó một điểm được đẩy lên ngăn xếp, ta hãy xét một tính chất hình học khác, được minh họa trong các Hình

35.9(a) và (b). Cho P là một đa giác lồi, và chọn một cạnh $\overline{p_r p_t}$ bất kỳ của P. Xét vùng được định cận bởi $\overline{p_r p_t}$ và các phần mở rộng của hai cạnh kề. (Tùy thuộc vào các góc tương đối của các cạnh kề, vùng có thể được định cận, như vùng tô bóng trong phần (a), hoặc không định cận, như vùng tô bóng trong phần (b).) Nếu ta bổ sung một điểm p_s bất kỳ trong vùng này vào P dưới dạng một đỉnh mới, có các cạnh $\overline{p_r p_s}$ và $\overline{p_s p_t}$ thay cạnh $\overline{p_r p_t}$, đa giác kết quả sẽ là lồi.

Giờ đây xét một điểm p_i được đẩy lên S. Tham khảo lại Hình 35.8(*b*), cho p_j là đỉnh trên đầu S ngay trước khi đẩy p_i, và cho p_k là phần tử tiền vị của p_j trên S. Ta biện luận rằng p_i phải rơi trong vùng tô bóng của Hình 35.8(*b*), tương ứng trực tiếp với các vùng tô bóng của Hình 35.9. Bởi góc $\angle p_k p_j p_i$ thực hiện một lần quay trái, nên p_i phải nằm trên phía tô bóng của phần mở rộng của $\overline{p_k p_j}$. Bởi p_i theo p_j trong cách sắp xếp góc cực, nên nó phải nằm trên phía tô bóng của $\overline{p_0 p_j}$. Hơn nữa, do cách mà ta đã chọn p_0, nên điểm p_i phải nằm trên phía tô bóng của phần mở rộng của $\overline{p_0 p_1}$. Như vậy, p_i nằm trong vùng tô bóng, và do đó sau khi p_i được đẩy lên ngăn xếp S, các điểm trên S hình thành một đa giác lồi. Điều này hoàn tất phần chứng minh của biện luận.

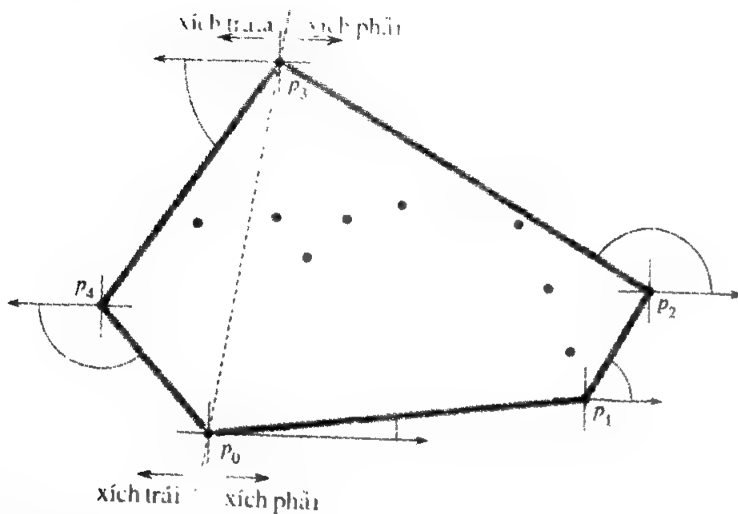
Do đó, vào cuối GRAHAM-SCAN, các điểm của Q nằm trên ngăn xếp S hình thành các đỉnh của một đa giác lồi. Ta đã chứng tỏ tất cả các điểm không nằm trên S đều không phải là các đỉnh của CH(Q) hoặc, theo tương đương, rằng tất cả các đỉnh của CH(Q) nằm trên S. Bởi S chỉ chứa các đỉnh từ Q và các điểm của nó hình thành một đa giác lồi, nên chúng phải hình thành CH(Q).



Hình 35.9 Bổ sung một điểm trong vùng tô bóng vào một đa giác lồi P cho ra một đa giác lồi khác. Vùng tô bóng được định cận bởi một cạnh của $\overline{p_r p_t}$ và các phần mở rộng của hai cạnh kề. **(a)** Vùng tô bóng được định cận. **(b)** Vùng tô bóng không được định cận.

Giờ đây ta chứng tỏ thời gian thực hiện của GRAHAM-SCAN là $O(n \lg n)$, ở đó $n = |Q|$. Dòng 1 mất $\Theta(n)$ thời gian. Dòng 2 mất $O(n \lg n)$ thời gian, dùng kỹ thuật sắp xếp trộn hoặc sắp xếp đống để sắp xếp các góc cực và phương pháp tích vectơ trong Đoạn 35.1 để so sánh các góc. (Có thể tiến hành gỡ bỏ tất cả ngoại trừ điểm xa nhất có cùng góc cực trong một tổng $O(n)$ thời gian.) Các dòng 3-6 mất $O(1)$ thời gian. Bởi $m \leq n - 1$, vòng lặp **for** của các dòng 7-10 được thi hành tối đa $n - 3$ lần. Bởi PUSH mất $O(1)$ thời gian, mỗi lần lặp lại mất $O(1)$ thời gian không kể thời gian bỏ ra trong vòng lặp **while** của các dòng 8-9, và như vậy vòng lặp **for** chung sẽ mất $O(n)$ thời gian loại trừ vòng lặp **while** được lồng.

Ta dùng phương pháp kết tập của phân tích khấu trừ để chứng tỏ vòng lặp **while** mất $O(n)$ thời gian nói chung. Với $i = 0, 1, ..., m$, mỗi điểm p_i được đẩy lên ngăn xếp S chính xác một lần. Như trong phân tích của thủ tục MULTIPOP trong Đoạn 18.1, ta nhận thấy có tối đa một phép toán POP cho mỗi phép toán PUSH. Ít nhất ba điểm—p_0, p_1, và p_m—không bao giờ được kéo ra khỏi ngăn xếp, sao cho thực tế có tối đa tổng cộng $m - 2$ phép toán POP được thực hiện. Mỗi lần lặp lại của vòng lặp **while** thực hiện một POP, và do đó có tối đa $m - 2$ lần lặp lại của vòng lặp **while**. Bởi đợt kiểm tra trong dòng 8 chiếm $O(1)$ thời gian, mỗi lệnh gọi POP sẽ mất $O(1)$ thời gian, và $m \leq n - 1$, tổng thời gian mà vòng lặp **while** chiếm là $O(n)$. Như vậy, thời gian thực hiện của GRAHAM-SCAN là $O(n \lg n)$.



Hình 35.10 Phép toán của điều hành Jarvis. Đỉnh đầu tiên được chọn là điểm thấp nhất p_0. Đỉnh kế tiếp, p_1, có góc cực bé nhất của một điểm bất kỳ đối với p_0. Như vậy, p_2 có góc cực bé nhất đối với p_1. Xích phải tiến cao tới mức điểm cao nhất p_3. Như vậy, xích trái được kiến tạo bằng cách tìm các góc cực bé nhất đối với trục x âm.

Diễu hành Jarvis

Diễu hành Jarvis tính toán bao lồi của một tập hợp Q các điểm bằng một kỹ thuật có tên ***đóng gói*** [package wrapping] (hoặc ***gói quà*** [gift wrapping]). Thuật toán chạy trong thời gian $O(nh)$, ở đó h là số lượng các đỉnh của CH(Q). Khi h là $o(\lg n)$, thuật toán diễu hành Jarvis theo tiệm cận nhanh hơn thuật toán quét Graham.

Theo trực giác, diễu hành Jarvis mô phỏng cách gói một mẩu giấy căng quanh tập hợp Q. Ta bắt đầu bằng cách buộc đầu tờ giấy vào điểm thấp nhất trong tập hợp, nghĩa là, vào cùng điểm p_0 mà ta bắt đầu đợt quét Graham với nó. Điểm này là một đỉnh của bao lồi. Ta kéo giấy về bên phải để làm nó căng, rồi ta kéo nó cao hơn cho đến khi nó chạm một điểm. Điểm này cũng phải là một đỉnh của bao lồi. Giữ giấy căng, ta tiếp tục theo cách này quanh tập hợp các đỉnh cho đến khi ta trở lại điểm p_0 ban đầu của chúng ta.

Chính thức hơn, thuật toán diễu hành Jarvis xây dựng một dãy $H = (p_0, p_1, ..., p_{h-1})$ gồm các đỉnh của CH(Q). Ta bắt đầu với p_0. Như Hình 35.10 đã nêu, đỉnh bao lồi kế tiếp p_1 có góc cực bé nhất đối với p_0. (Trong trường hợp các mối ràng buộc, ta chọn điểm xa nhất từ p_0.) Cũng vậy, p_2 có góc cực bé nhất đối với p_1, vân vân. Khi ta đạt đến đỉnh cao nhất, giả sử p_k (ngắt các mối ràng buộc bằng cách chọn đỉnh xa nhất như vậy), ta đã kiến tạo ***xích phải*** [right chain] của CH (Q), như Hình 35.10 đã nêu. Để kiến tạo ***xích trái*** [left chain], ta bắt đầu tại p_k và chọn p_{k+1} với tư cách là điểm có góc cực bé nhất đối với p_k, nhưng *từ trục x âm.* Cứ thế tiếp tục, hình thành xích trái bằng cách lấy các góc cực từ trục x âm, cho đến khi ta trở lại đỉnh p_0 ban đầu.

Ta có thể thực thi thuật toán diễu hành Jarvis trong một đợt quét khái niệm quanh bao lồi, nghĩa là, không kiến tạo riêng biệt các xích phải và trái. Các thực thi như vậy thường theo dõi góc của canh bao lồi cuối được chọn và yêu cầu dãy các góc của các cạnh bao tăng ngặt (trong miền giá trị từ 0 đến 2π radian). Ưu điểm của việc kiến tạo các xích riêng biệt đó là không cần tính toán rõ rệt các góc; các kỹ thuật trong Đoạn 35.1 là đủ để so sánh các góc.

Nếu được thực thi đúng đắn, diễu hành Jarvis có một thời gian thực hiện là $O(nh)$. Với mỗi trong số h đỉnh của CH(Q), ta tìm ra đỉnh có góc cực cực tiểu. Mỗi phép so sánh giữa các góc cực sẽ mất $O(1)$ thời gian, dùng các kỹ thuật của Đoạn 35.1. Như Đoạn 10.1 đã nêu, ta có thể tính toán cực tiểu của n giá trị trong $O(n)$ thời gian nếu mỗi phép so sánh mất $O(1)$ thời gian. Như vậy, diễu hành Jarvis chiếm $O(nh)$ thời gian.

Bài tập

35.3-1

Chứng minh trong thủ tục GRAHAM-SCAN, các điểm p_1 và p_m phải là các đỉnh của CH(Q).

35.3-2

Xét một mô hình tính toán hỗ trợ phép cộng, phép so sánh, và phép nhân và với nó ta có một cận dưới là $\Omega(n \lg n)$ để sắp xếp n con số. Chứng minh $\Omega(n \lg n)$ là một cận dưới để tính toán, theo thứ tự, các đỉnh của bao lồi của một tập hợp n điểm trong một mô hình như vậy.

35.3-3

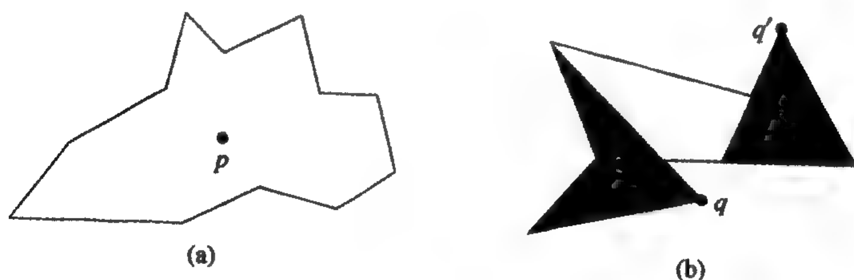
Cho một tập hợp các điểm Q, chứng minh cặp điểm xa nhau nhất phải là các đỉnh của CH(Q).

35.3-4

Với một đa giác P đã cho và một điểm q trên biên của nó, ***bóng*** của q là tập hợp các điểm r sao cho đoạn thẳng $\overline{qr}$ hoàn toàn nằm trên biên hoặc trong phần trong của P. Một đa giác ***P có hình sao*** nếu ở đó tồn tại một điểm p trong phần trong của P nằm trong bóng của mọi điểm trên biên của P. Tập hợp tất cả các điểm p như vậy được gọi là ***lõi*** của P. (Xem Hình 35.11.) Cho một đa giác P n-đỉnh, hình sao được chỉ định bởi các đỉnh của nó theo thứ tự ngược chiều kim đồng hồ, hãy nêu cách tính toán CH(P) trong $O(n)$ thời gian.

35.3-5

Trong ***bài toán bao lồi trực tuyến***, ta có tập hợp Q gồm n điểm, lần lượt từng điểm một. Sau khi nhận từng điểm, ta phải tính toán bao lồi



Hình 35.11 Phần định nghĩa của một đa giác hình sao, để dùng trong Bài tập 35.3-4. **(a)** Một đa giác hình sao. Đoạn thẳng từ điểm p đến một điểm q bất kỳ trên biên chỉ cắt biên tại q. **(b)** Một đa giác phi hình sao. Vùng tô bóng bên trái là bóng của q, và vùng tô bóng bên phải là bóng của q'. Bởi các vùng này rời nhau, nên lõi trống rỗng.

các điểm đã thấy cho đến giờ. Hiển nhiên, ta có thể chạy thuật toán quét Graham mỗi lần một điểm, với một tổng thời gian thực hiện là $O(n^2 \lg n)$. Nêu cách giải bài toán bao lồi trực tuyến trong một tổng $O(n^2)$ thời gian.

35.3-6 *

Nêu cách thực thi phương pháp gia số để tính toán bao lồi n điểm sao cho nó chạy trong $O(n \lg n)$ thời gian.

35.4 Tìm cặp điểm sát nhất

Giờ đây, ta xét bài toán tìm the cặp điểm sát nhất trong một tập hợp Q gồm $n \geq 2$ điểm. "Sát nhất" có nghĩa là khoảng cách euclid bình thường: khoảng cách giữa các điểm $p_1 = (x_1, y_1)$ và $p_2 = (x_2, y)$ là $\sqrt{(x_1 - x_2)^2 - (y_1 - y_2)^2}$. Hai điểm trong tập hợp Q có thể là trùng khớp, trong trường hợp đó khoảng cách giữa chúng là zero. Ví dụ, bài toán này có các ứng dụng trong các hệ thống điều khiển lưu thông. Một hệ thống để điều khiển lưu thông trên không hoặc trên biển có thể cần biết hai chiếc tàu nào là sát nhất để phát hiện các va chạm tiềm ẩn.

Một thuật toán cặp sát nhất cưỡng bức đơn giản xem xét tất cả = $\Theta(n^2)$ cặp điểm. Trong đoạn này, ta mô tả một thuật toán chia để trị cho bài toán này mà thời gian thực hiện của nó được mô tả bằng phép truy toán quen thuộc $T(n) = 2T(n/2)+O(n)$. Như vậy, thuật toán này chỉ sử dụng $O(n \lg n)$ thời gian.

Thuật toán chia để trị

Mỗi lần triệu gọi đệ quy của thuật toán nhận một tập hợp con $P \subseteq Q$ và các mảng X và Y làm đầu vào, mỗi mảng chứa tất cả các điểm của tập hợp con đầu vào P. Các điểm trong mảng X được sắp xếp sao cho các tọa độ x của chúng tăng đơn điệu. Cũng vậy, mảng Y được sắp xếp bằng cách tăng đơn điệu tọa độ y. Lưu ý, để đạt được cận thời gian $O(n \lg n)$, ta không tài nào có đủ khả năng để sắp xếp trong từng lệnh gọi đệ quy; nếu làm thế, phép truy toán cho thời gian thực hiện sẽ là $T(n) = 2T(n/2) + O(n \lg n)$, mà nghiệm của nó là $T(n) = O(n \lg^2 n)$. Ta sẽ thấy cách sử dụng phép "sắp xếp trước" để duy trì tính chất đã sắp xếp này mà không thực tế sắp xếp trong mỗi lệnh gọi đệ quy.

Một lần triệu gọi đệ quy đã cho có các đầu vào P, X, và Y trước tiên sẽ kiểm tra xem $|P| \leq 3$ hay không. Nếu có, lệnh triệu gọi đơn giản thực hiện phương pháp cưỡng bức mô tả trên đây: thử tất cả $\binom{|P|}{2}$ cặp

điểm và trả về cặp sát nhất. Nếu $|P| > 3$, lệnh triệu gọi đệ quy thi hành mô thức chia để trị như sau.

Chia: Nó tìm một vạch dọc l cắt đôi tập hợp điểm P thành hai tập hợp P_L và P_R sao cho $|P_L| = \lceil |P|/2 \rceil$, $|P_R| = \lfloor |P|/2 \rfloor$, tất cả các điểm trong P_L nằm trên hoặc về bên trái của vạch l, và tất cả các điểm trong P_R nằm trên hoặc về bên phải của l. Mảng X được chia thành các mảng X_L và X_R, chứa các điểm của P_L và P_R theo thứ tự nêu trên, được sắp xếp bằng cách tăng đơn điệu tọa độ x. Cũng vậy, mảng Y được chia thành các mảng Y_L và Y_R, chứa các điểm của P_L và P_R theo thứ tự nêu trên, được sắp xếp bằng cách tăng đơn điệu tọa độ y.

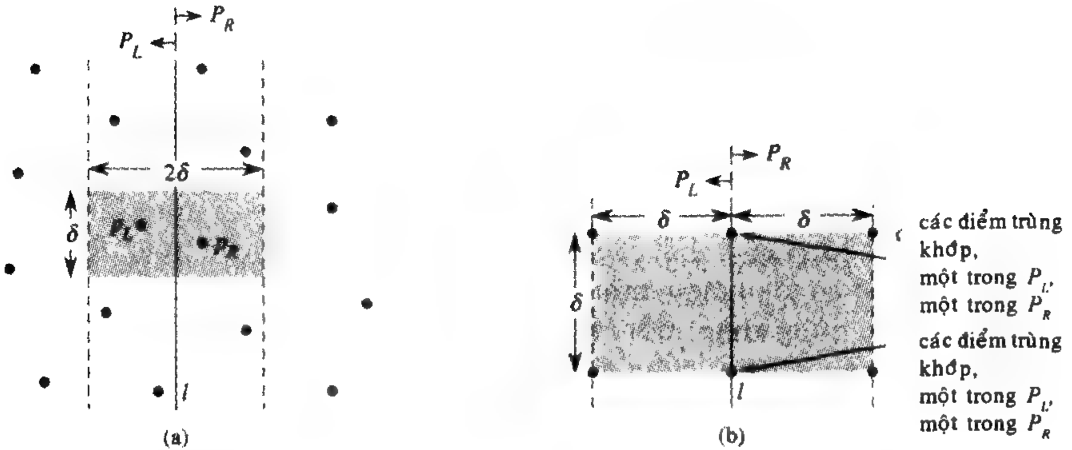
Trị: Chia P thành P_L và P_R, nó thực hiện hai lệnh gọi đệ quy, một để tìm cặp điểm sát nhất trong P_L và một để tìm cặp điểm sát nhất trong P_R. Các đầu vào cho lệnh gọi đầu tiên là tập hợp con P_L và các mảng X_L và Y_L; lệnh gọi thứ hai nhận các đầu vào P_R, X_R, và Y_R. Cho các khoảng cách cặp sát nhất trả về cho P_L và P_R là δ_L và δ_R, theo thứ tự nêu trên, và cho $\delta = \min(\delta_L, \delta_R)$.

Tổ hợp: Cặp sát nhất là cặp có khoảng cách δ mà một trong các lệnh gọi đệ quy tìm thấy, hoặc nó là một cặp điểm có một điểm trong P_L và điểm kia trong P_R. Thuật toán xác định xem có một cặp như vậy có khoảng cách nhỏ hơn δ hay không. Nhận thấy nếu có một cặp điểm có khoảng cách nhỏ hơn δ, cả hai điểm của cặp phải nằm trong δ đơn vị của vạch l. Như vậy, như Hình 35.12(a) đã nêu, cả hai phải thường trú trong sọc dọc rộng 2δ được căn giữa tại vạch l. Để tìm một cặp như vậy, nếu có, thuật toán thực hiện như sau.

1. Nó tạo một mảng Y', là mảng Y có tất cả các điểm không nằm trong sọc dọc rộng 2δ được gỡ bỏ. Mảng Y' được sắp xếp bởi tọa độ y, giống như Y.

2. Với mỗi điểm p trong mảng Y', thuật toán gắng tìm các điểm trong Y' nằm trong δ đơn vị của p. Như sẽ thấy dưới đây, chỉ 7 điểm trong Y' theo p mới cần được xét. Thuật toán tính toán khoảng cách từ p đến từng trong số 7 điểm này và theo dõi khoảng cách cặp sát nhất δ' tìm thấy trên tất cả các cặp điểm trong Y'.

3. Nếu $\delta' < \delta$, thì sọc dọc quả thực chứa một cặp sát hơn cặp mà các lệnh gọi đệ quy đã tìm thấy. Cặp này và khoảng cách δ' của nó được trả về. Bằng không, cặp sát nhất và khoảng cách δ của nó mà các lệnh gọi đệ quy tìm thấy sẽ được trả về.



Hình 35.12 Các khái niệm chính trong phần chứng minh mà thuật toán cặp sát nhất cần kiểm tra chỉ 7 điểm theo sau mỗi điểm trong mảng *Y*'. **(a)** Nếu $p_L \in P_L$ và p_R *E* P_R nhỏ hơn δ đơn vị tách rời, chúng phải thường trú trong một hình chữ nhật $\delta \times 2\delta$ được căn giữa tại vạch *l*. **(b)** Tất cả 4 điểm theo từng cặp ít nhất δ đơn vị tách rời có thể thường trú ra sao trong một hình vuông $\delta \times \delta$. Bên trái là 4 điểm trong P_L, và bên phải là 4 điểm trong P_R. Có thể có δ điểm trong hình chữ nhật $\delta \times 2\delta$ nếu các điểm được nêu trên vạch *l* thực tế là các cặp điểm trùng khớp có một điểm trong P_L và một điểm trong P_R.

Phần mô tả trên đây bỏ qua vài chi tiết thực thi cần có để đạt được $O(n \lg n)$ thời gian thực hiện. Sau khi chứng minh tính đúng đắn của thuật toán, ta sẽ nêu cách thực thi thuật toán để đạt được cận thời gian mong muốn.

Tính đúng đắn

Tính đúng đắn của thuật toán cặp sát nhất này là hiển nhiên, ngoại trừ hai khía cạnh. Thứ nhất, bằng cách hạ đệ quy xuống đến mức thấp nhất khi $|P| \leq 3$, ta bảo đảm không bao giờ gắng chia một tập hợp các điểm với chỉ một điểm. Khía cạnh thứ hai đó là cần chỉ kiểm tra 7 điểm theo mỗi điểm *p* trong mảng *Y*'; giờ đây ta sẽ chứng minh tính chất này.

Giả sử tại một cấp đệ quy nào đó, cặp điểm sát nhất là $p_L \in P_L$ và $p_R \in P_R$. Như vậy, khoảng cách δ giữa p_L và p_R hoàn toàn nhỏ hơn δ. Điểm p_L phải nằm trên hoặc về bên trái của vạch *l* và nhỏ hơn δ đơn vị tách rời. Cũng vậy, p_R nằm trên hoặc về bên phải của *l* và nhỏ hơn δ đơn vị tách rời. Hơn nữa, p_L và p_R nằm trong δ đơn vị với nhau theo chiều dọc. Như vậy, như Hình 35.12**(a)** đã nêu, p_L và p_R nằm trong một hình chữ nhật $\delta \times 2\delta$ được căn giữa tại vạch *l*. (Cũng có thể có các điểm khác trong hình chữ nhật này.)

Kế đó, ta chứng tỏ có tối đa δ điểm của *P* có thể thường trú trong

hình chữ nhật $\delta \times 2\delta$ này. Xét hình vuông $\delta \times \delta$ hình thành nửa trái của hình chữ nhật này. Bởi tất cả các điểm trong P_L ít nhất là δ đơn vị tách rời, nên có tối đa 4 điểm có thể thường trú trong hình vuông này; Hình 35.12(b) có nêu cách. Cũng vậy, có tối đa 4 điểm trong P_R có thể thường trú trong hình vuông $\delta \times \delta$ hình thành nửa phải của hình chữ nhật. Như vậy, có tối đa 8 điểm của P có thể thường trú trong hình chữ nhật $\delta \times 2\delta$. (Lưu ý, bởi các điểm trên vạch l có thể nằm trong P_L hoặc P_R, có thể có tới 4 điểm trên l. Có thể đạt được giới hạn này nếu có hai cặp điểm trùng khớp, mỗi cặp bao gồm một điểm từ P_L và một điểm từ P_R, một cặp nằm tại điểm giao của l và đỉnh của hình chữ nhật, và cặp kia là nơi l cắt đáy của hình chữ nhật.)

Sau khi chứng tỏ có tối đa 8 điểm của P có thể thường trú trong hình chữ nhật, ta dễ dàng thấy chỉ cần chỉ kiểm tra 7 điểm theo sau mỗi điểm trong mảng Y'. Vẫn giả định cặp sát nhất là p_L và p_R, ta hãy mặc nhận mà không để mất tính tổng quát rằng p_L đứng trước p_R trong mảng Y'. Như vậy, cho dù p_L xảy ra càng sớm càng tốt trong Y' và p_R xảy ra càng trễ càng tốt, p_R vẫn nằm trong một trong số 7 vị trí theo sau p_L. Như vậy, ta đã chứng tỏ tính đúng đắn của thuật toán cặp sát nhất.

Cách thực thi và thời gian thực hiện

Như đã lưu ý, mục tiêu của chúng ta đó là có phép truy toán với thời gian thực hiện là $T(n) = 2T(n/2) + O(n)$, ở đó tất nhiên, $T(n)$ là thời gian thực hiện cho một tập hợp n điểm. Khó khăn chính đó là bảo đảm các mảng X_L, X_R, Y_L, và Y_R, chuyền cho các lệnh gọi đệ quy, sẽ được sắp xếp bởi đúng tọa độ và cũng bảo đảm rằng mảng Y' được sắp xếp bởi tọa độ y. (Lưu ý, nếu mảng X mà một lệnh gọi đệ quy nhận đã được sắp xếp sẵn, thì ta dễ dàng hoàn tất phép chia tập hợp P thành P_L và P_R trong thời gian tuyến tính.)

Nhận xét chính đó là trong mỗi lệnh gọi, ta muốn hình thành một tập hợp con đã sắp xếp của một mảng đã sắp xếp. Ví dụ, một lần triệu gọi cụ thể được gán tập hợp con P và mảng Y, được sắp xếp bởi tọa độ y. Sau khi đã phân hoạch P thành P_L và P_R, nó cần hình thành các mảng Y_L và Y_R, được sắp xếp bởi tọa độ y. Hơn nữa, các mảng này phải được hình thành trong thời gian tuyến tính. Phương pháp có thể được xem là đối lập với thủ tục MERGE từ kỹ thuật sắp xếp trộn trong Đoạn 1.3.1: ta đang tách một mảng đã sắp xếp thành hai mảng đã sắp xếp. Mã giả dưới đây cho ta ý tưởng.

```
1 length[Y_L] ← length[Y_R] ← 0
2 for i ← 1 to length[Y]
3       do if Y[i] ∈ P_L
```

```
        then length[Y_L] ← length[Y_L] + 1
                Y[length[Y_L]] ← Y[i]
        else length[Y_R] ← length[Y_R] + 1
                Y[length[Y_R]] ← Y[i]
```

Ta đơn giản xét các điểm trong mảng Y theo thứ tự. Nếu một điểm $Y[i]$ nằm trong P_L, ta chắp nó vào cuối mảng Y_L; bằng không, ta chắp nó vào cuối mảng Y_R. Mã giả tương tự làm việc để hình thành các mảng X_L, X_R, và Y'.

Câu hỏi duy nhất còn lại đó là cách có các điểm đã sắp xếp đầu tiên. Ta thực hiện điều này bằng cách đơn giản ***sắp xếp trước*** chúng; nghĩa là, ta sắp xếp chúng một lần cho tất cả *trước* lệnh gọi đệ quy đầu tiên. Các mảng đã sắp xếp được chuyền vào lệnh gọi đệ quy đầu tiên, và từ đó chúng được bớt dần qua các lệnh gọi đệ quy khi cần. Tính năng sắp xếp trước bổ sung thêm một $O(n \lg n)$ vào thời gian thực hiện, nhưng giờ đây mỗi bước của đệ quy chiếm thời gian tuyến tính ngoại trừ các lệnh gọi đệ quy. Như vậy, nếu ta cho $T(n)$ là thời gian thực hiện của mỗi bước đệ quy và $T'(n)$ là thời gian thực hiện của nguyên cả thuật toán, ta có $T'(n) = T(n) + O(n \lg n)$ và

$$T(n) = \begin{cases} 2T(n/2) + O(n) & \text{nếu } n > 3\,, \\ O(1) & \text{nếu } n \le 3\,. \end{cases}$$

Như vậy, $T(n) = O(n \lg n)$ và $T'(n) = O(n \lg n)$.

Bài tập

35.4-1

Giáo sư Smothers nảy ra một lược đồ cho phép thuật toán cặp sát nhất kiểm tra chỉ 5 điểm theo sau mỗi điểm trong mảng Y'. Ý tưởng đó là luôn đặt các điểm trên vạch l vào tập hợp P_L. Như vậy, không thể có các cặp điểm trùng khớp trên vạch l có một điểm trong P_L và một điểm trong P_R. Như vậy, có tối đa 6 điểm có thể thường trú trong hình chữ nhật $\delta \times 2\delta$. Đâu là điểm sai lầm trong lược đồ của giáo sư?

35.4-2

Không cần tăng thời gian thực hiện tiệm cận của thuật toán, hãy nêu cách bảo đảm tập hợp các điểm được chuyền cho lệnh gọi đệ quy đầu tiên không chứa các điểm trùng khớp. Chứng minh rằng như vậy ta chỉ cần kiểm tra các điểm trong 6 (chứ không phải 7) vị trí mảng theo sau mỗi điểm trong mảng Y'. Tại sao lại không đủ khi chỉ kiểm tra có 5 vị trí mảng theo sau mỗi điểm?

35.4-3

Khoảng cách giữa hai điểm có thể được định nghĩa theo các cách khác ngaòi euclid. Trong mặt phẳng, ***khoảng cách*** L_m- giữa các điểm p_1 và p_2 được căn cứ vào $((x_1 - x_2)^m + (y_1 - y_2)^m)^{1/m}$. Do đó, khoảng cách Euclid là khoảng cách L_2. Sửa đổi thuật toán cặp sát nhất để sử dụng khoảng cách L_1- còn gọi là ***khoảng cách Manhattan.***

35.4-4

Cho hai điểm p_1 và p_2 trong mặt phẳng, khoảng cách L_∞ - giữa chúng là $\max(|x_1 - x_2|, |y_1 - y_2|)$. Sửa đổi thuật toán cặp sát nhất để sử dụng khoảng cách L_∞.

Các Bài Toán

35-1 Các tầng lồi

Cho một tập hợp Q gồm các điểm trong mặt phẳng, ta định nghĩa các ***tầng lồi*** của Q theo quy nạp. Tầng lồi đầu tiên của Q bao gồm các điểm trong Q là các đỉnh của CH(Q). Với $i > 1$, định nghĩa Q_i để bao gồm các điểm của Q với tất cả các điểm trong các tầng lồi 1, 2,..., i -1 được gỡ bỏ. Như vậy, tầng lồi thứ i của Q là CH(Q_i) nếu $Q_i \neq \emptyset$ và bằng không là không xác định.

a. Nêu một thuật toán $O(n^2)$ thời gian để tìm các tầng lồi của một tập hợp trên n điểm.

b. Chứng minh cần có $\Omega(n \lg n)$ thời gian để tính toán các tầng lồi của một tập hợp n điểm trên một mô hình tính toán bất kỳ yêu cầu $\Omega(n \lg n)$ thời gian để sắp xếp n số thực.

35-2 Các tầng tột bậc

Cho Q là một tập hợp n điểm trong mặt phẳng. Ta nói rằng điểm (x, y) ***chi phối*** điểm (x', y') nếu $x \geq x'$ và $y \geq y'$. Một điểm trong Q không bị chi phối bởi các điểm khác trong Q được gọi là ***tột bậc*** [maximal]. Lưu ý, Q có thể chứa nhiều điểm tột bậc, có thể được tổ chức thành các ***tầng tột bậc*** như sau. Tầng tột bậc đầu tiên L_1 là tập hợp các điểm tột bậc của Q. Với $i > 1$, tầng tột bậc thứ i L_i là tập hợp các điểm tột bậc trong $Q - \bigcup_{j=1}^{i-1} L_j$.

Giả sử Q có k tầng tột bậc không trống, và cho y_i là tọa độ y của điểm mút trái trong L_i với $i = 1, 2, ..., k$. Trước mắt, mặc nhận rằng không có hai điểm trong Q có cùng tọa độ x hoặc y.

a. Chứng tỏ $y_1 > y_2 > ... > y_k$.

Xét một điểm (x, y) về bên trái của một điểm bất kỳ trong Q và với nó y là riêng biệt từ tọa độ y của một điểm bất kỳ trong Q. Cho $Q' = Q \cup \{(x, y)\}$.

b. Cho j là chỉ số cực tiểu sao cho $y_j < y$, trừ phi $y < y_k$, trong trường hợp đó ta cho $j = k + 1$. Chứng tỏ các tầng tột bậc của Q' là như sau.

- Nếu $j \leq k$, thì các tầng tột bậc của Q' giống như các tầng tột bậc của Q, ngoại trừ L_j cũng gộp (x, y) làm điểm mút trái mới của nó.

- Nếu $j = k + 1$, thì k tầng tột bậc đầu tiên của Q' giống như của Q, nhưng ngoài ra, Q' có một tầng tột bậc thứ $(k + 1)$ không trống: $L_{k+1} = \{(x, y)\}$.

c. Mô tả một thuật toán $O(n \lg n)$ thời gian để tính toán các tầng tột bậc của một tập hợp Q n điểm. (*Mách nước*: Dời một đường quét từ phải qua trái.)

d. Có những điểm khác biệt nào nảy sinh nếu giờ đây ta cho phép các điểm đầu vào có cùng tọa độ x hoặc y? Gợi ý một cách để giải các bài toán như vậy.

35-3 Các thầy pháp và con ma

Một nhóm n thầy pháp đang đánh với n con ma. Mỗi thầy pháp được trang bị một gói proton, bắn ra một luồng vào một con ma, trừ tiệt nó. Một luồng đi theo một đường thẳng và kết thúc khi nó ném trúng con ma. Các thầy pháp quyết định dựa trên chiến lược sau. Chúng sẽ bắt cặp với các con ma, hình thành n cặp thầy pháp-con ma, rồi đồng thời mỗi thầy pháp sẽ bắn một luồng vào con ma được chọn của họ. Như tất cả chúng ta đều biết, quả *rất* nguy hiểm khi để các luồng đi qua, và do đó các thầy pháp phải chọn các kiểu bắt cặp để không có luồng nào sẽ đi qua.

Giả sử vị trí của mỗi thầy pháp và mỗi con ma là một điểm cố định trong mặt phẳng và không có ba vị trí nào là cộng tuyến.

a. Chứng tỏ ở đó tồn tại một đường thẳng đi qua một thầy pháp và một con ma số lượng các thầy pháp như vậy trên một phía của đường thẳng bằng số lượng các con ma trên cùng phía. Mô tả cách tìm một đường thẳng như vậy trong $O(n \lg n)$ thời gian.

b. Nêu một thuật toán $O(n^2 \lg n)$ thời gian để bắt cặp các thầy pháp với các con ma sao cho không có luồng nào đi qua.

35-4 *Các phép phân phối có bao thưa*

Xét bài toán tính toán bao lồi của một tập hợp các điểm trong mặt

phẳng đã được rút theo một phép phân phối ngẫu nhiên đã biết nào đó. Đôi lúc, bao lồi n điểm được rút từ một phép phân phối như vậy có kích cỡ dự trù $O(n^{1-\epsilon})$ với một hằng $\varepsilon > 0$. Ta gọi một phép phân phối như vậy là có ***bao thưa*** [sparse-hulled]. Các phép phân phối có bao thưa bao gồm:

- Các điểm được rút đồng đều từ một đĩa có bán kính đơn vị. Bao lồi có kích cỡ dự trù $\Theta(n^{1/3})$.

- Các điểm được rút đồng đều từ phần trong của một đa giác lồi có k cạnh, với một hằng k. Bao lồi có kích cỡ dự trù $\Theta(\lg n)$.

- Các điểm được rút theo một phép phân phối bình thường hai chiều. Bao lồi có kích cỡ dự trù $\Theta(\sqrt{\lg n})$.

a. Cho hai đa giác lồi có n_1 và n_2 đỉnh theo thứ tự nêu trên, nêu cách tính toán bao lồi của tất cả $n_1 + n_2$ điểm trong $O(n_1 + n_2)$ thời gian. (Các đa giác có thể phủ chồng.)

b. Chứng tỏ bao lồi của một tập hợp n điểm được rút độc lập theo một phép phân phối có bao thưa có thể được tính toán trong $O(n)$ thời gian dự trù. (*Mách nước:* Theo đệ quy tìm ra các bao lồi của $n/2$ điểm đầu tiên và $n/2$ điểm thứ hai, rồi tổ hợp các kết quả.)

Ghi chú Chương

Chương này chỉ mới đề cập khái quát về các thuật toán và các kỹ thuật hình học điện toán. Các sách nói về hình học điện toán bao gồm của Preparata và Shamos [160] và Edelsbrunner [60].

Tuy hình học đã được nghiên cứu bởi người thời xưa, song sự phát triển của các thuật toán cho các bài toán hình học là tương đối mới. Preparata và Shamos lưu ý rằng khái niệm sớm nhất về tính tính phức tạp của một bài toán đã được E. Lemoine cung cấp vào năm 1902. Ông đã nghiên cứu các kiến tạo euclid—là các kiến tạo dùng một thước và một cạnh bẹt [straightedge]—và đã nghĩ ra một tập hợp năm nguyên hàm: đặt một chân của compa trên một điểm đã cho, đặt một chân của compa trên một đường đã cho, vẽ một vòng tròn, chuyền cạnh của thước qua một điểm đã cho, và vẽ một đường. Lemoine quan tâm đến số lượng các nguyên hàm cần có để đem lại một kiến tạo đã cho; ông gọi lượng này là "tính đơn giản" của kiến tạo.

Thuật toán của Đoạn 35.2, xác định các đoạn thẳng bất kỳ có giao nhau hay không, là của Shamos và Hoey [176].

Phiên bản ban đầu của quét Graham là do Graham [91] cung cấp. Thuật toán đóng gói là của Jarvis [112]. Dùng một mô hình tính toán cây-quyết định, Yao [205] đã chứng minh một cận dưới của $\Omega(n \lg n)$ cho thời gian thực hiện của bất kỳ thuật toán bao lồi nào. Khi số lượng các đỉnh h của bao lồi được tính, thuật toán lược tìm Kirkpatrick và Seidel [120], mất $O(n \lg h)$ thời gian, sẽ là tối ưu theo tiệm cận.

Thuật toán $O(n \lg n)$ thời gian chia để trị để tìm cặp điểm sát nhất là của Shamos và xuất hiện trong Preparata và Shamos [160]. Preparata và Shamos cũng chứng tỏ thuật toán là tối ưu theo tiệm cận trong một mô hình cây-quyết định.

36 Tính Đầy Đủ NP

Tất cả của các thuật toán mà ta đã nghiên cứu cho đến giờ đều là các ***thuật toán thời gian đa thức***: trên các đầu vào có kích cỡ n, thời gian thực hiện trường hợp xấu nhất của chúng là $O(n^k)$ với một hằng k. Quả rất tự nhiên khi ta tự hỏi rằng có phải *tất cả* các bài toán đều có thể giải trong thời gian đa thức. Câu trả lời là không. Ví dụ, có các bài toán, như "Bài toán treo" [Halting Problem] nổi tiếng của Turing không thể giải bởi bất kỳ máy tính nào, bất kể cung cấp bao nhiêu thời gian. Cũng có các bài toán có thể giải được, nhưng không phải trong thời gian $O(n^k)$ với một hằng k. Nói chung, ta xem các bài toán có thể giải được bằng các thuật toán thời gian đa thức là "dễ trị", và các bài toán yêu cầu thời gian siêu đa thức là "khó trị".

Tuy nhiên, chủ đề của chương này là một lớp bài toán đáng quan tâm, có tên các bài toán"đầy đủ NP", mà tình trạng của nó là ẩn số. Chưa có thuật toán thời gian đa thức nào được khám phá cho bài toán đầy đủ NP, cũng như chưa có ai có thể chứng minh một cận dưới thời gian siêu đa thức với bất kỳ trong số chúng. Cái gọi là vấn đề $P \neq NP$ này đã từng là một trong các bài toán nghiên cứu mở gây phức tạp nhất, sâu nhất, trong khoa học máy tính lý thuyết bởi nó đã được đặt ra vào năm 1971.

Hầu hết các nhà khoa học máy tính lý thuyết đều tin rằng các bài toán đầy đủ NP đều khó trị. Lý do đó là nếu có thể giải bất kỳ bài toán đầy đủ NP đơn lẻ nào trong thời gian đa thức, thì *mọi* bài toán đầy đủ NP sẽ có một thuật toán thời gian đa thức. Căn cứ vào phạm vi rộng các bài toán đầy đủ NP đã được nghiên cứu cho đến nay, mà không có sự tiến bộ nào về phía một giải pháp thời gian đa thức, ta thực sự ngạc nhiên nếu như có thể giải tất cả chúng trong thời gian đa thức.

Để trở thành một nhà thiết kế thuật toán tốt, bạn phải hiểu các khái niệm cơ bản của lý thuyết về tính đầy đủ NP [NP-completeness]. Nếu có thể thiết lập một bài toán dưới dạng đầy đủ NP, bạn cung cấp một bằng chứng tốt về tính khó trị của nó. Là một kỹ sư, tốt hơn bạn nên bỏ thời gian phát triển một thuật toán xấp xỉ (xem Chương 37) thay vì tìm kiếm một thuật toán nhanh để giải bài toán một cách chính xác. Hơn

nữa, nhiều bài toán tự nhiên và đáng quan tâm mà bề ngoài dường như không khó hơn bài toán sắp xếp, tìm đồ thị, hoặc luồng mạng lại thực tế mang tính đầy đủ NP. Như vậy, điều quan trọng đó là bạn nên quen thuộc với lớp bài toán đáng lưu ý này.

Chương này nghiên cứu các khía cạnh về tính đầy đủ NP có liên quan trực tiếp nhất đến việc phân tích của các thuật toán. Trong Đoạn 36.1, ta hình thức hóa khái niệm về "bài toán" và định nghĩa lớp P phức tạp gồm các bài toán quyết định có thể giải trong thời gian đa thức. Ta cũng xem cách phối hợp các khái niệm này vào khuôn khổ lý thuyết ngôn ngữ hình thức. Đoạn 36.2 định nghĩa lớp NP gồm các bài toán quyết định mà các nghiệm của chúng có thể được xác minh trong thời gian đa thức. Nó cũng chính thức đặt vấn đề P ≠ NP.

Đoạn 36.3 nêu cách nghiên cứu các mối quan hệ giữa các bài toán thông qua các kiểu "rút gọn" thời gian đa thức. Nó định nghĩa tính đầy đủ NP và phác thảo một chứng minh rằng một bài toán, có "khả năng thỏa mãn mạch" [circuit satisfiability] là đầy đủ NP. Sau khi tìm thấy một bài toán đầy đủ NP, Đoạn 36.4 sẽ cho thấy cách chứng minh các bài toán khác là đầy đủ NP sẽ trở nên đơn giản hơn nhiều bằng phương pháp luận về các phép rút gọn [reductions]. Phương pháp luận được minh họa bằng cách chứng tỏ hai bài toán về khả năng thỏa công thức là đầy đủ NP. Nhiều bài toán khác được chứng tỏ là đầy đủ NP trong Đoạn 36.5.

36.1 Thời gian đa thức

Dn bắt đầu phần nghiên cứu về tính đầy đủ NP, ta hình thức hóa khái niệm về các bài toán giải được theo thời gian đa thức. Các bài toán này thường được xem là "dễ trị." Lý do tại sao là một vấn đề về triết học, chứ không phải về toán học. Ta có thể đưa ra ba lý lẽ hỗ trợ.

Thứ nhất, mặc dù có lý khi xem bài toán yêu cầu thời gian $\Theta(n^{100})$ là khó trị, song có rất ít bài toán thực tiễn yêu cầu thời gian dựa trên thứ tự một đa thức độ cao như vậy. Các bài toán tính toán được thời gian đa thức đã gặp trong thực tế thường yêu cầu ít thời gian hơn nhiều.

Thứ hai, Với nhiều mô hình tính toán hợp lý, một bài toán có thể giải trong thời gian đa thức theo một mô hình có thể được giải trong thời gian đa thức theo một mô hình khác. Ví dụ, lớp bài toán giải được trong thời gian đa thức bằng máy truy cập ngẫu nhiên nối tiếp được dùng xuyên suốt cuốn sách này giống như lớp bài toán giải được trong thời gian đa

thức trên các máy Turing trừu tượng.[1] Nó cũng giống như lớp bài toán giải được trong thời gian đa thức trên một máy tính song song, cho dù số lượng các bộ xử lý tăng trưởng theo đa thức với kích cỡ đầu vào.

Thứ ba, lớp bài toán thời gian đa thức giải được có các tính chất bao đóng chính xác, bởi các đa thức được đóng dưới phép cộng, phép nhân, và phép hợp. Ví dụ, nếu kết xuất của một thuật toán thời gian đa thức được nạp vào đầu vào của một thuật toán khác, thuật toán phức hợp sẽ là đa thức. Nếu một thuật toán thời gian đa thức khác thực hiện một số lệnh gọi không đổi đến các chương trình con thời gian đa thức, thời gian thực hiện của thuật toán phức hợp là đa thức.

Các bài toán trừu tượng

Để hiểu rõ lớp các bài toán giải được theo thời gian đa thức, trước tiên ta phải có một khái niệm chính thức về "bài toán" [problem] là gì. Ta định nghĩa một ***bài toán trừu tượng Q*** là một hệ thức nhị phân trên một tập hợp *I* các ***minh dụ*** [instances] bài toán và một tập hợp *S* các ***nghiệm*** bài toán. Ví dụ, xét bài toán SHORTEST-PATH tìm một lộ trình ngắn nhất giữa hai đỉnh đã cho trong một đồ thị không gia trọng không hướng $G = (V, E)$. Một minh dụ cho SHORTEST-PATH là một bộ ba bao gồm một đồ thị và hai đỉnh. Một nghiệm là một dãy các đỉnh trong đồ thị, với có lẽ dãy trống biểu hiện không có lộ trình nào tồn tại. Chính bài toán SHORTEST-PATH là hệ thức kết hợp từng minh dụ của một đồ thị và hai đỉnh với một lộ trình ngắn nhất trong đồ thị nối hai đỉnh. Bởi các lộ trình ngắn nhất không nhất thiết là duy nhất, nên một minh dụ bài toán đã cho có thể có nhiều nghiệm.

Cách trình bày này của một bài toán trừu tượng là chung hơn so với yêu cầu của các mục tiêu của chúng ta. Để đơn giản, lý thuyết về tính đầy đủ NP hạn chế sự chú ý vào các ***bài toán quyết định***: có một nghiệm có/không. Trong trường hợp này, ta có thể xem một bài toán quyết định trừu tượng dưới dạng một hàm ánh xạ tập hợp minh dụ *I* theo tập hợp nghiệm $\{0,1\}$. Ví dụ, một bài toán quyết định PATH liên quan đến bài toán lộ trình ngắn nhất là, "Cho một đồ thị $G = (V, E)$, hai đỉnh $u, v \in V$, và một số nguyên không âm *k*, hỏi có tồn tại một lộ trình trong *G* giữa *u* và *v* mà chiều dài của nó tối đa là *k* hay không?" Nếu $i = \langle G, u, v, k \rangle$ là một minh dụ của bài toán lộ trình ngắn nhất này, thì $\text{PATH}(i) = 1$ (có) nếu một lộ trình ngắn nhất từ *u* đến *v* có chiều dài tối đa *k*, và bằng không $\text{PATH}(i) = 0$ (không).

Nhiều bài toán trừu tượng không phải là bài toán quyết định, mà là

---

[1] Xem Hopcroft và Ullman [104] hoặc Lewis và Papadimitriou [139] để có phần nghiên cứu kỹ về mô hình máy Turing.

các ***bài toán tối ưu hóa***, ở đó một giá trị nào đó phải được giảm thiểu hoặc tối đa hóa. Để áp dụng lý thuyết về tính đầy đủ NP cho các bài toán tối ưu hóa, ta phải áp đổi lại chúng thành các bài toán quyết định. Thông thường, để áp đổi lại một bài toán tối ưu hóa, ta áp đặt một cận trên giá trị sẽ được tối ưu hóa. Để lấy ví dụ, trong khi áp đổi lại bài toán lộ trình ngắn nhất thành bài toán quyết định PATH, ta đã bổ sung một cận k vào minh dụ bài toán.

Mặc dù lý thuyết về tính đầy đủ NP buộc ta áp đổi lại các bài toán tối ưu hóa thành các bài toán quyết định, yêu cầu này không giảm bớt sự tác động của lý thuyết. Nói chung, nếu có thể nhanh chóng giải một bài toán tối ưu hóa, ta cũng có thể nhanh chóng giải bài toán quyết định có liên quan của nó. Ta đơn giản so sánh giá trị có được từ nghiệm của bài toán tối ưu hóa với cận được cung cấp làm đầu vào cho bài toán quyết định. Do đó, nếu một bài toán tối ưu hóa là dễ, bài toán quyết định có liên quan của nó cũng dễ. Để phát biểu theo một cách có liên quan hơn đến tính đầy đủ NP, nếu ta có thể cung cấp bằng chứng cho thấy một bài toán quyết định là khó, ta cũng cung cấp bằng chứng bài toán tối ưu hóa có liên quan của nó là khó. Như vậy, cho dù nó hạn chế sự chú ý vào các bài toán quyết định, lý thuyết về tính đầy đủ NP vẫn áp dụng rộng rãi hơn nhiều.

Các phép mã hóa

Nếu mục tiêu của một chương trình máy tính là giải một bài toán trừu tượng, các minh dụ bài toán phải được biểu diễn theo cách mà chương trình hiểu rõ. ***Phép mã hóa*** [encoding] một tập hợp S các đối tượng trừu tượng chính là phép ánh xạ e từ S theo tập hợp các chuỗi nhị phân.[2] Ví dụ, tất cả chúng ta đều quen thuộc với phép mã hóa các số tự nhiên $\mathbf{N} = \{0, 1, 2, 3, 4, ...\}$ dưới dạng các chuỗi $\{0, 1, 10, 11, 100, ...\}$. Dùng phép mã hóa này, $e(17) = 10001$. Những ai đã xem xét các phép biểu diễn máy tính về các ký tự trên bàn phím ắt đã quen thuộc với các mã ASCII hoặc EBCDIC. Trong mã ASCII, $e(\text{A}) = 1000001$. Thậm chí một đối tượng phức hợp có thể được mã hóa dưới dạng một chuỗi nhị phân bằng cách tổ hợp các phép biểu diễn của các thành phần cấu thành của nó. Các đa giác, đồ thị, các hàm, các cặp có thứ tự, các chương trình—tất cả đều có thể được mã hóa dưới dạng các chuỗi nhị phân.

Như vậy, một thuật toán máy tính "giải" một bài toán quyết định trừu tượng nào đó thực tế chấp nhận một phép mã hóa của một minh dụ bài toán làm đầu vào. Ta gọi một bài toán có tập hợp minh dụ là tập hợp các

---

[2] Đồng miền xác định của e không cần phải là các chuỗi *nhị phân*; bất kỳ tập hợp các chuỗi trên một bảng chữ cái hữu hạn có ít nhất 2 ký hiệu đều được.

chuỗi nhị phân là một ***bài toán cụ thể***. Ta nói rằng một thuật toán ***giải*** một bài toán cụ thể trong thời gian $O(T(n))$ nếu, khi được cung cấp một minh dụ bài toán i có chiều dài $n = |i|$, thuật toán có thể tạo ra nghiệm trong thời gian tối đa $O(T(n))$. Do đó, một bài toán cụ thể là ***giải được theo thời gian đa thức***, nếu ở đó tồn tại một thuật toán để giải nó trong thời gian $O(n^k)$ với một hằng k.

Giờ đây, ta có thể chính thức định nghĩa ***lớp phức* P** dưới dạng tập hợp các bài toán quyết định cụ thể giải được trong thời gian đa thức.

Ta có thể dùng các phép mã hóa để ánh xạ các bài toán trừu tượng theo các bài toán cụ thể. Cho một bài toán quyết định trừu tượng Q ánh xạ một tập hợp minh dụ I theo $\{0, 1\}$, ta có thể dùng một phép mã hóa $e : I \to \{0, 1\}^*$ để cảm sinh một bài toán quyết định cụ thể có liên quan, mà ta thể hiện bằng $e(Q)$. Nếu nghiệm cho một minh dụ bài toán trừu tượng $i \in I$ là $Q(i) \in \{0, 1\}$, thì nghiệm cho minh dụ bài toán cụ thể $e(i) \in \{0, 1\}^*$ cũng là $Q(i)$. Với tư cách là một tính chất kỹ thuật, có thể có vài chuỗi nhị phân không biểu diễn một bài toán minh dụ trừu tượng có ý nghĩa. Để tiện dụng, ta mặc nhận rằng mọi chuỗi như vậy được ánh xạ theo 0 một cách tùy ý. Như vậy, bài toán cụ thể tạo ra cùng các nghiệm như bài toán trừu tượng trên các minh dụ chuỗi nhị phân biểu thị cho các phép mã hóa của các minh dụ bài toán trừu tượng.

Ta muốn mở rộng phần định nghĩa về khả năng giải thời gian đa thức từ các bài toán cụ thể sang các bài toán trừu tượng dùng các phép mã hóa làm cầu nối, nhưng ta muốn phần định nghĩa độc lập với bất kỳ phép mã hóa cụ thể nào. Nghĩa là, hiệu năng về việc giải một bài toán không được tùy thuộc vào cách mã hóa bài toán. Đáng tiếc, nó lại tùy thuộc khá nặng. Ví dụ, giả sử một số nguyên k phải được cung cấp dưới dạng đầu vào duy nhất cho một thuật toán, và giả sử thời gian thực hiện của thuật toán là $\Theta(k)$. Nếu số nguyên k được cung cấp theo ***đơn phân*** [unary]—một chuỗi gồm k 1—thì thời gian thực hiện của thuật toán là $O(n)$ trên các đầu vào có chiều dài-n, là thời gian đa thức. Tuy nhiên, nếu ta dùng phần biểu diễn nhị phân tự nhiên hơn của số nguyên k, thì chiều dài đầu vào là $n = \lceil \lg k \rceil$. Trong trường hợp này, thời gian thực hiện của thuật toán là $\Theta(k) = \Theta(2^n)$, là hàm số mũ theo kích cỡ của đầu vào. Như vậy, tùy thuộc vào phép mã hóa, thuật toán chạy trong thời gian đa thức hay siêu đa thức.

Do đó, phép mã hóa của một bài toán trừu tượng là khá quan trọng đối với sự hiểu biết của chúng ta về thời gian đa thức. Ta không thể thực sự nói về cách giải một bài toán trừu tượng mà trước tiên không chỉ định một phép mã hóa. Tuy vậy, trong thực tế, nếu ta loại trừ các phép mã hóa "tốn kém" như mã hóa đơn phân, phép mã hóa thực tế

của một bài toán chẳng mấy quan trọng đối với việc bài toán có thể giải trong thời gian đa thức hay không. Ví dụ, việc biểu thị các số nguyên theo cơ số 3 thay vì nhị phân sẽ không có tác động gì với việc một bài toán có giải được trong thời gian đa thức hay không, bởi một số nguyên được biểu diễn theo cơ số 3 có thể được chuyển đổi thành một số nguyên được biểu diễn theo cơ số 2 trong thời gian đa thức.

Ta nói rằng một hàm $f : \{0, 1\}^* \to \{0,1\}^*$ là ***tính toán được theo thời gian đa thức*** nếu ở đó tồn tại một thuật toán thời gian đa thức A mà, căn cứ vào một đầu vào $x \in \{0,1\}^*$, sẽ tạo ra $f(x)$ dưới dạng kết xuất. Với một tập hợp I các minh dụ bài toán, ta nói rằng hai phép mã hóa e_1 và e_2 là ***có liên quan theo đa thức*** nếu ở đó tồn tại hai hàm tính toán được theo thời gian đa thức f_{12} và f_{21} sao cho với mọi $i \in I$, ta có $f_{12}(e_1(i)) = e_2(i)$ và $f_{21}(e_2(i)) = e_1(i)$. Nghĩa là, phép mã hóa $e_2(i)$ có thể được tính toán từ phép mã hóa $e_1(i)$ bằng một thuật toán thời gian đa thức, và ngược lại. Nếu hai phép mã hóa e_1 và e_2 của một bài toán trừu tượng có liên quan theo đa thức, mà ta sử dụng, không quan trọng gì đến việc bài toán có giải được theo thời gian đa thức hay không, như bổ đề dưới đây cho thấy.

Bổ đề 36.1

Cho Q là một bài toán quyết định trừu tượng trên một tập hợp minh dụ I, và cho e_1 và e_2 là các phép mã hóa có liên quan theo đa thức trên I. Thì, $e_1(Q) \in \mathrm{P}$ nếu và chỉ nếu $e_2(Q) \in \mathrm{P}$.

Chứng minh Ta chỉ cần chứng minh hướng tới, bởi hướng lui là đối xứng. Do đó, giả sử $e_1(Q)$ có thể giải trong thời gian $O(n^k)$ với một hằng k. Hơn nữa, giả sử với mọi minh dụ bài toán i, phép mã hóa $e_1(i)$ có thể được tính toán từ phép mã hóa $e_2(i)$ trong thời gian $O(n^c)$ với một hằng c, ở đó $n = |e_1(i)|$. Để giải bài toán $e_2(Q)$, trên đầu vào $e_2(i)$, trước tiên ta tính toán $e_1(i)$ rồi chạy thuật toán với $e_1(Q)$ trên $e_1(i)$. Điều này kéo dài bao lâu? Việc chuyển đổi các phép mã hóa mất một thời gian $O(n^c)$, và do đó $|e_1(i)| = O(n^c)$, bởi kết xuất của một máy tính nối tiếp không thể dài hơn thời gian thực hiện của nó. Việc giải bài toán trên $e_1(i)$ mất một thời gian $O(|e_1(i)|^k) = O(n^{ck})$, là đa thức bởi cả c lẫn k đều là các hằng.

Như vậy, dẫu một bài toán trừu tượng có các minh dụ của nó được mã hóa theo nhị phân hay cơ số 3, điều đó không ảnh hưởng đến "tính phức hợp" của nó, nghĩa là, dẫu nó có giải được theo thời gian đa thức hay không, nhưng nếu các minh dụ được mã hóa theo đơn phân, tính phức hợp của nó vẫn có thể thay đổi. Để có thể đảo đề theo kiểu độc lập với phép mã hóa, ta thường mặc nhận các minh dụ bài toán được mã hóa theo một kiểu hợp lý, súc tích, trừ phi ta cụ thể nói khác đi. Để

chính xác, ta mặc nhận rằng phép mã hóa của một số nguyên có liên quan theo đa thức đến phần biểu diễn nhị phân của nó, và phép mã hóa của một tập hợp hữu hạn có liên quan theo đa thức đến phép mã hóa của nó dưới dạng một danh sách các thành phần của nó, được bao trong các dấu ngoặc ôm và được tách biệt bởi các dấu phẩy. (ASCII là một lược đồ mã hóa như vậy.) Với một phép mã hóa "chuẩn" như vậy trong tay, ta có thể suy ra các phép mã hóa hợp lý của các đối tượng toán học khác, như các tuple, đồ thị, và các công thức. Để thể hiện phép mã hóa chuẩn của một đối tượng, ta sẽ bao đối tượng trong các dấu ngoặc nhọn. Như vậy, $<G>$ thể hiện phép mã hóa chuẩn của một đồ thị G.

Miễn là mặc định dùng một phép mã hóa có liên quan theo đa thức đến phép mã hóa chuẩn này, ta có thể nói trực tiếp về các bài toán trừu tượng mà không cần tham chiếu bất kỳ phép mã hóa cụ thể nào, biết rằng việc chọn lựa phép mã hóa không ảnh hưởng gì đến việc bài toán trừu tượng có giải được theo thời gian đa thức hay không. Từ đây trở đi, nói chung ta sẽ mặc nhận rằng tất cả các minh dụ bài toán là các chuỗi nhị phân được mã hóa bằng phép mã hóa chuẩn, trừ phi ta rõ rệt chỉ định ngược lại. Ta cũng thường bỏ qua sự phân biệt giữa các bài toán trừu tượng và cụ thể. Tuy nhiên, độc giả nên chú ý về các bài toán nảy sinh trong thực tế, ở đó một phép mã hóa chuẩn không hiển nhiên và phép mã hóa là quan trọng.

Một khuôn khổ ngôn ngữ hình thức

Một trong các khía cạnh tiện dụng của việc tập trung vào các bài toán quyết định đó là chúng giúp ta dễ dàng sử dụng máy móc của lý thuyết ngôn ngữ hình thức. Ở đây ta cũng nên ôn lại vài định nghĩa từ lý thuyết này. Một ***bảng chữ cái*** Σ là một tập hợp hữu hạn các ký hiệu. Một ***ngôn ngữ*** L trên Σ là một tập hợp các chuỗi bất kỳ được hình thành bởi các ký hiệu từ Σ. Ví dụ, nếu $\Sigma = \{0, 1\}$, tập hợp $L = \{10, 11, 101, 111, 1011, 1101, 10001, ...\}$ là ngôn ngữ của các phần biểu diễn nhị phân của các số nguyên tố. Ta ký hiệu ***chuỗi trống*** là ε, và ***ngôn ngữ trống*** là $\emptyset$. Ngôn ngữ của tất cả các chuỗi trên Σ được ký hiệu là Σ^*. Ví dụ, nếu $\Sigma = \{0, 1\}$, thì $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, ...\}$ là tập hợp tất cả các chuỗi nhị phân. Mọi ngôn ngữ L trên Σ là một tập hợp con của Σ^*.

Có nhiều phép toán trên các ngôn ngữ. Các phép toán lý thuyết tập hợp, như ***hợp*** và ***giao***, trực tiếp đến từ các định nghĩa về lý thuyết tập hợp. Ta định nghĩa ***phép bù*** của L bằng $\overline{L} = \Sigma^* - L$. ***Phần ghép nối*** của hai ngôn ngữ L_1 và L_2 là ngôn ngữ

$$L = \{x_1 x_2 : x_1 \in L_1 \text{ và } x_2 \in L_2\}.$$

Bao đóng hoặc ***sao Kleene*** của một ngôn ngữ L là ngôn ngữ

$L^* = \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cup \ldots$,

ở đó L^k là ngôn ngữ có được bằng cách ghép nối L với chính nó k lần.

Từ quan điểm của lý thuyết ngôn ngữ, tập hợp các minh dụ cho một bài toán quyết định Q bất kỳ đơn giản là tập hợp Σ^*, ở đó $\Sigma = \{0,1\}$. Bởi Q hoàn toàn được biểu thị bởi các minh dụ bài toán tạo ra một đáp án 1 (có), nên ta có thể xem Q như một ngôn ngữ L trên $\Sigma = \{0, 1\}$, ở đó

$L = \{x \in \Sigma^* : Q(x) = 1\}$.

Ví dụ, bài toán quyết định PATH có ngôn ngữ tương ứng

PATH = {<G, u, v, k> : $G = (V, E)$ là một đồ thị không hướng,
$u, v \in V$,
$k \geq 0$ là một số nguyên, và
ở đó tồn tại một lộ trình từ u đến v trong G
mà chiều dài của nó tối đa là k} .

(Để tiện dụng, đôi lúc ta dùng cùng tên—PATH trong trường hợp này—để tham chiếu cả bài toán quyết định lẫn ngôn ngữ tương ứng của nó.)

Khuôn khổ ngôn ngữ hình thức cho phép ta diễn tả hệ thức giữa các bài toán quyết định và các thuật toán giải chúng một cách súc tích. Ta nói rằng một thuật toán A ***chấp nhận*** một chuỗi $x \in \{0, 1\}^*$ nếu, căn cứ vào đầu vào x, thuật toán kết xuất $A(x) = 1$. Ngôn ngữ được một thuật toán A ***chấp nhận*** là tập hợp $L = \{x \in \{0, 1\}^* : A(x) = 1\}$, nghĩa là, tập hợp các chuỗi mà thuật toán chấp nhận. Một thuật toán A ***loại bỏ*** một chuỗi x nếu $A(x) = 0$.

Cho dù ngôn ngữ L được chấp nhận bởi một thuật toán A, song thuật toán sẽ không nhất thiết loại bỏ một chuỗi $x \notin L$ được cung cấp dưới dạng đầu vào cho nó. Ví dụ, thuật toán có thể lặp vòng mãi. Một ngôn ngữ L được ***quyết định*** bởi một thuật toán A nếu mọi chuỗi nhị phân được thuật toán chấp nhận hoặc loại bỏ. Một ngôn ngữ L được ***chấp nhận trong thời gian đa thức*** bằng một thuật toán A nếu với bất kỳ chuỗi $x \in L$ có chiều dài-n, thuật toán chấp nhận x trong thời gian $O(n^k)$ với một hằng k. Một ngôn ngữ ***L được quyết định trong thời gian đa thức*** bởi một thuật toán A nếu với bất kỳ chuỗi $x \in \{0,1\}^*$ có chiều dài-n, thuật toán quyết định x trong thời gian $O(n^k)$ với một hằng k. Như vậy, để chấp nhận một ngôn ngữ, một thuật toán chỉ cần quan tâm về các chuỗi trong L, nhưng để quyết định một ngôn ngữ, nó phải chấp nhận hoặc loại bỏ mọi chuỗi trong $\{0, 1\}^*$.

Để lấy ví dụ, ngôn ngữ PATH có thể được chấp nhận trong thời gian đa thức. Một thuật toán chấp nhận thời gian đa thức sẽ tính toán lộ trình ngắn nhất từ u đến v trong G, dùng kỹ thuật tìm kiếm độ rộng đầu tiên, rồi so sánh khoảng cách có được với k. Nếu khoảng cách tối đa là k, thuật toán kết xuất 1 và treo. Bằng không, thuật toán chạy mãi. Tuy nhiên, thuật toán này không decide PATH, bởi nó không dứt khoát kết xuất 0 cho các minh dụ ở đó lộ trình ngắn nhất có chiều dài lớn hơn k. Một thuật toán quyết định cho PATH phải rõ rệt loại bỏ các chuỗi nhị phân không thuộc về PATH. Với một bài toán quyết định như PATH, một thuật toán quyết định như vậy thường dễ thiết kế. Với các bài toán khác, như Bài toán Treo của Turing, ở đó tồn tại một thuật toán chấp nhận, nhưng không có thuật toán quyết định nào tồn tại.

Một cách không chính thức, ta có thể định nghĩa một ***lớp phức*** [complexity class] dưới dạng một tập hợp các ngôn ngữ, tư cách thành viên ở đó được xác định bởi một ***độ đo phức*** [complexity measure], như thời gian thực hiện, trên một thuật toán xác định xem một chuỗi đã cho x có thuộc về ngôn ngữ L hay không. Phần định nghĩa thực tế về một lớp phức có phần kỹ thuật hơn—độc giả có quan tâm có thể tham khảo tài liệu của Hartmanis và Steams [95].

Dùng khuôn khổ lý thuyết ngôn ngữ này, ta có thể cung cấp một định nghĩa khác của lớp phức P:

P = $\{L \subseteq \{0, 1\}^*$: ở đó tồn tại một thuật toán A

quyết định L trong thời gian đa thức$\}$.

Thực vậy, P cũng là lớp các ngôn ngữ có thể được chấp nhận trong thời gian đa thức.

Định lý 36.2

P = $\{L$: L được chấp nhận bởi một thuật toán thời gian đa thức$\}$.

Chứng minh Bởi lớp ngôn ngữ mà các thuật toán thời gian đa thức quyết định là một tập hợp con của lớp ngôn ngữ được chấp nhận bởi các thuật toán thời gian đa thức, nên ta chỉ cần chứng tỏ rằng nếu L được chấp nhận bởi một thuật toán thời gian đa thức, nó được quyết định bởi một thuật toán thời gian đa thức. Cho L là ngôn ngữ được chấp nhận bởi một thuật toán thời gian đa thức A. Ta sẽ dùng một lập luận "mô phỏng" cổ điển để kiến tạo một thuật toán thời gian đa thức A' khác quyết định L. Bởi A chấp nhận L trong thời gian $O(n^k)$ với một hằng k, nên ở đó cũng tồn tại một hằng c sao cho A chấp nhận L trong tối đa $T = cn^k$ bước. Với một chuỗi đầu vào x, thuật toán A' mô phỏng hành động của A trong thời gian T. Vào cuối thời gian T, thuật toán A' thẩm tra cách ứng xử của A. Nếu A đã chấp nhận x, thì A' chấp nhận x bằng cách

kết xuất một 1. Nếu A không chấp nhận x, thì A' loại bỏ x bằng cách kết xuất một 0. Tổng phí của A' mô phỏng A không gia tăng thời gian thực hiện lên trên mức một thừa số đa thức, và như vậy A' là một thuật toán thời gian đa thức quyết định L.

Lưu ý, phần chứng minh của Định lý 36.2 là phi kiến tạo. Với một ngôn ngữ đã cho $L \in$ P, thực tế ta có thể không biết một cận trên thời gian thực hiện cho thuật toán A chấp nhận L. Tuy vậy, ta biết rằng một cận như vậy tồn tại, và do đó, một thuật toán A' tồn tại có thể kiểm tra cận, cho dù ta không thể dễ dàng tìm ra thuật toán A'.

Bài tập

36.1-1

Định nghĩa bài toán tối ưu hóa LONGEST-PATH-LENGTH dưới dạng hệ thức kết hợp từng minh dụ của một đồ thị không hướng và hai đỉnh với chiều dài của lộ trình đơn giản dài nhất giữa hai đỉnh. Định nghĩa bài toán quyết định LONGEST-PATH = $\{<G, u, v, k> : G = (V, E)$ là một đồ thị không hướng, $u, v \in V$, $k \geq 0$ là một số nguyên, và ở đó tồn tại một lộ trình đơn giản từ u đến v trong G có chiều dài ít nhất là $k\}$. Chứng tỏ bài toán tối ưu hóa LONGEST-PATH-LENGTH có thể được giải trong thời gian đa thức nếu và chỉ nếu LONGEST-PATH $\in$ P.

36.1-2

Nêu một định nghĩa hình thức cho bài toán tìm chu trình đơn giản dài nhất trong một đồ thị không hướng. Nêu một bài toán quyết định có liên quan. Nêu ngôn ngữ tương ứng với bài toán quyết định.

36.1-3

Nêu một phép mã hóa chính thức của đồ thị có hướng dưới dạng các chuỗi nhị phân dùng một phép biểu diễn ma trận kề. Thực hiện như vậy dùng một phép biểu diễn danh sách kề. Chứng tỏ hai phép biểu diễn đều có có liên quan theo đa thức.

36.1-4

Thuật toán lập trình động cho bài toán ba lô 0-1 mà Bài tập 17.2-2 yêu cầu có phải là một thuật toán thời gian đa thức không? Giải thích đáp án của bạn.

36.1-5

Giả sử một ngôn ngữ L có thể chấp nhận một chuỗi $x \in L$ trong thời gian đa thức, nhưng giả sử thuật toán thực hiện điều này lại chạy trong thời gian siêu đa thức nếu $x \notin L$. Chứng tỏ L có thể được

quyết định trong thời gian đa thức.

36.1-6

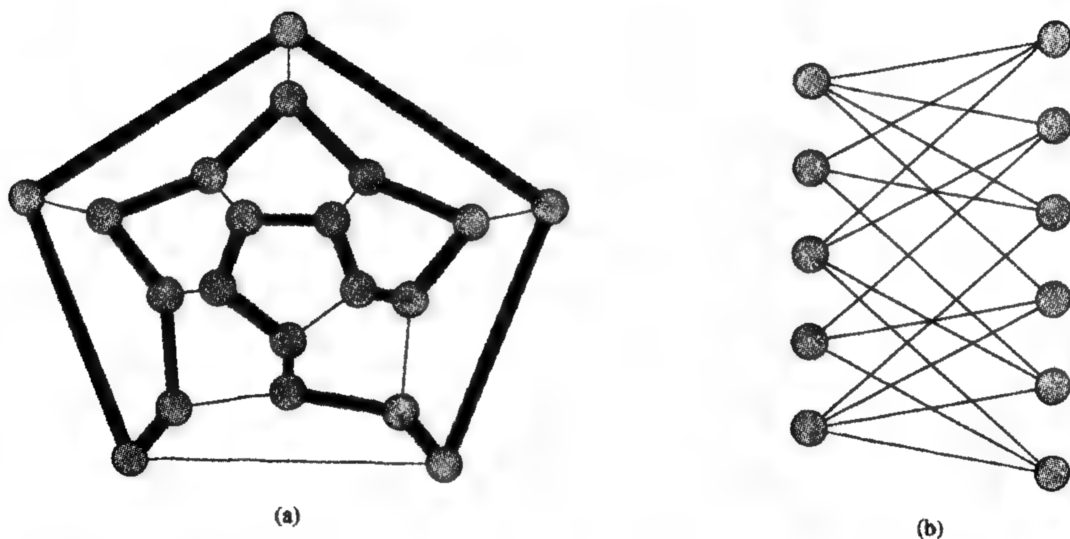
Chứng tỏ rằng một thuật toán thực hiện tối đa một số lệnh gọi không đổi đến các chương trình con thời gian đa thức sẽ chạy trong thời gian đa thức, nhưng một số lệnh gọi đa thức đến các chương trình con thời gian đa thức có thể dẫn đến một thuật toán thời gian mũ.

36.1-7

Chứng tỏ lớp P, được xem là một tập hợp các ngôn ngữ, được đóng dưới phép hợp, giao, ghép nối , bù, và sao Kleene. Nghĩa là, nếu L_1, L_2 $\in$ P, thì $L_1 \cup L_2 \in$ P, vân vân.

36.2 Xác minh thời gian đa thức

Giờ đây, ta xét các thuật toán "xác minh" tư cách thành viên trong các ngôn ngữ. Ví dụ, giả sử rằng với một minh dụ đã cho <*G*, *u*, *v*, *k*> của bài toán quyết định PATH, ta cũng có một lộ trình *p* từ *u* đến *v*. Ta có thể dễ dàng kiểm tra chiều dài của *p* có phải tối đa là *k* hay không, và nếu có, ta có thể xem *p* như một "chứng chỉ" rằng minh dụ quả thực thuộc về PATH. Với bài toán quyết định PATH, chứng chỉ dường như không làm ta quan tâm lắm. Xét cho cùng, PATH thuộc về P—thực tế, PATH có thể được giải trong thời gian tuyến tính—và do đó việc xác minh tư cách thành viên từ một chứng chỉ đã cho sẽ kéo dài ngang bằng



Hình 36.1 (a) Một đồ thị biểu thị các đỉnh, các cạnh, và các mặt của một khối mười hai mặt có một chu trình hamilton được nêu bằng các cạnh tô bóng. **(b)** Một đồ thị hai nhánh có một số đỉnh lẻ. Mọi kiểu đồ thị như vậy đều là phi hamilton.

với việc giải bài toán từ đầu. Giờ đây, ta xét một bài toán mà ta biết chưa có một thuật toán quyết định thời gian đa thức, căn cứ vào một chứng chỉ, việc xác minh không khó.

Các chu trình hamilton

Bài toán tìm một chu trình hamilton trong một đồ thị không hướng đã được nghiên cứu trên trăm năm nay. Chính thức, một ***chu trình hamilton*** của một đồ thị không hướng $G = (V, E)$ là một chu trình đơn giản chứa mỗi đỉnh trong V. Một đồ thị chứa một chu trình hamilton được gọi là ***hamilton***; bằng không, là ***phi hamilton***. Bondy và Murty [31] trích dẫn một lá thư của W. R. Hamilton mô tả một trò chơi toán học trên khối mười hai mặt (Hình 36.1(a)) ở đó một người chơi cắm năm kim trong năm đỉnh liền kề bất kỳ và một người chơi khác phải hoàn thành lộ trình để tạo thành một chu trình chứa tất cả các đỉnh. Khối mười hai mặt là hamilton, và Hình 36.1 (a) nêu một chu trình hamilton. Tuy nhiên, không phải tất cả mọi đồ thị đều là hamilton. Ví dụ, Hình 36.1 (b) nêu một đồ thị hai nhánh có một số đỉnh lẻ. (Bài tập 36.2-2 yêu cầu bạn chứng tỏ tất cả các đồ thị như vậy đều là phi hamilton.)

Ta có thể định nghĩa ***bài toán chu trình hamilton***, "Một đồ thị G có một chu trình hamilton hay không?" dưới dạng một ngôn ngữ hình thức:

HAM-CYCLE = $\{\langle G\rangle : G$ là một đồ thị hamilton$\}$.

Một thuật toán có thể quyết định ngôn ngữ HAM-CYCLE bằng cách nào? Căn cứ vào một minh dụ bài toán $\langle G\rangle$, một thuật toán quyết định khả dĩ sẽ liệt kê tất cả các phép hoán vị của các đỉnh của G rồi kiểm tra từng phép hoán vị để xem nó có phải là một lộ trình hamilton hay không. Đâu là thời gian thực hiện của thuật toán này? Nếu ta dùng phép mã hóa "hợp lý" của một đồ thị dưới dạng ma trận kề của nó, số m đỉnh trong đồ thị là $\Omega(\sqrt{n})$, ở đó $n = |\langle G\rangle|$ là chiều dài của phép mã hóa của G. Có $m!$ phép hoán vị khả dĩ của các đỉnh, và do đó thời gian thực hiện là $\Omega(m!) = \Omega(\sqrt{n}!) = \Omega(2^{\sqrt{n}})$, không phải là $O(n^k)$ với một hằng k. Như vậy, thuật toán đơn sơ này không chạy trong thời gian đa thức, và thực tế, bài toán chu trình hamilton là đầy đủ NP, như sẽ chứng minh trong Đoạn 36.5.

Các thuật toán xác minh

Tuy nhiên, xét một bài toán có phần dễ hơn. Giả sử rằng một người bạn cho bạn biết một đồ thị G đã cho là hamilton, rồi xung phong chứng minh nó bằng cách cho bạn các đỉnh theo thứ tự dọc theo chu trình hamilton. Chắc chắn chẳng khó gì để xác minh phần chứng minh: chỉ việc xác minh chu trình đã cung cấp là hamilton bằng cách kiểm tra nó có phải là một phép hoán vị của các đỉnh của V và mỗi trong số các

cạnh liền kề dọc theo chu trình có thực tế tồn tại trong đồ thị hay không. Chắc chắn thuật toán xác minh này có thể được thực thi để chạy trong $O(n^2)$ thời gian, ở đó n là chiều dài của phép mã hóa của G. Như vậy, một phép chứng minh rằng một chu trình hamilton tồn tại trong một đồ thị có thể được xác minh trong thời gian đa thức.

Ta định nghĩa một ***thuật toán xác minh*** dưới dạng là một thuật toán hai đối số A, ở đó một đối số là một chuỗi đầu vào bình thường x và đối số kia là một chuỗi nhị phân y có tên một ***chứng chỉ*** [certificate]. Một thuật toán hai-đối số A ***xác minh*** một chuỗi đầu vào x nếu ở đó tồn tại một chứng chỉ y sao cho sao cho $A(x, y) = 1$. ***Ngôn ngữ được xác minh*** bởi một thuật toán xác minh A là

$$L = \{x \in \{0,1\}^* : \text{ở đó tồn tại } y \in \{0,1\}^* \text{ sao cho } A(x, y) = 1\} .$$

Theo trực giác, một thuật toán A xác minh một ngôn ngữ L xem với một chuỗi $x \in L$, ta có một chứng chỉ y mà A có thể dùng để chứng minh rằng $x \notin L$ hay không. Hơn nữa, với một chuỗi $x \in L$, không được có chứng chỉ nào chứng minh rằng $x \in L$. Ví dụ, trong bài toán chu trình hamilton, chứng chỉ là danh sách các đỉnh trong chu trình hamilton. Nếu một đồ thị là hamilton, bản thân chu trình hamilton sẽ cung cấp đủ thông tin để xác minh sự việc này. Ngược lại, nếu một đồ thị không phải là hamilton, thì không có danh sách các đỉnh nào có thể đánh lừa thuật toán xác minh tin rằng đồ thị là hamilton, bởi thuật toán xác minh kiểm tra cẩn thận "chu trình" đề xuất để bảo đảm chắc chắn.

Lớp phức NP

Lớp phức **NP** [complexity class NP] là lớp các ngôn ngữ có thể được xác minh bởi một thuật toán thời gian đa thức.[3] Chính xác hơn, một ngôn ngữ L thuộc về NP nếu và chỉ nếu ở đó tồn tại một thuật toán thời gian đa thức hai-đầu vào A và hằng c sao cho

$$L = \{x \in \{0,1\}^* : \text{ở đó tồn tại một chứng chỉ } y \text{ với } |y| = O(|x|^c) \text{ sao cho } A(x, y) = 1\} .$$

---

[3] Tên "NP" thay cho "nondeterministic polynomial time" [thời gian đa thức không tất định]. Thoạt đầu, lớp NP được nghiên cứu trong ngữ cảnh thuyết không tất định, nhưng cuốn sách này sử dụng khái niệm tuy hơi đơn giản hơn song vẫn tương đương về xác minh. Hopcroft và Ullman [104] có phần trình bày chi tiết về tính đầy đủ NP theo dạng các mô hình tính toán không tất định.

Ta nói rằng thuật toán A ***xác minh*** ngôn ngữ L ***trong thời gian đa thức.***

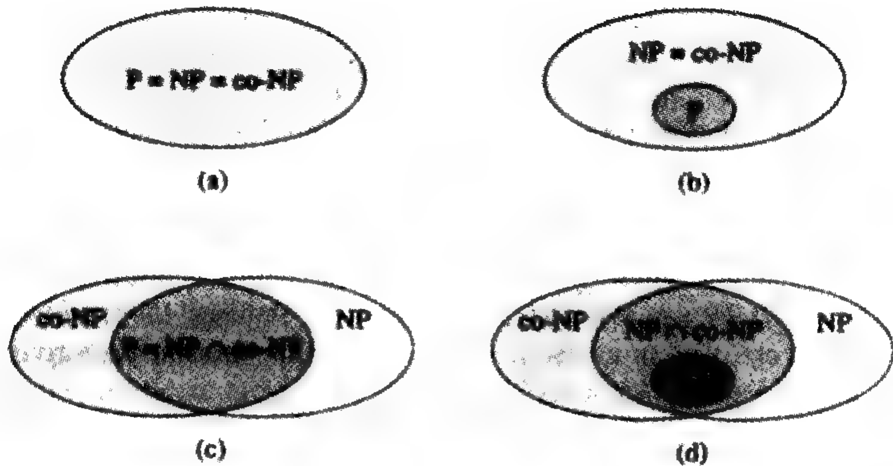
Từ phần thảo luận trên đây về bài toán chu trình hamilton, dẫn đến HAM-CYCLE ∈ NP. (Nên biết rằng một tập hợp quan trong là không trống.) Hơn nữa, nếu $L \in$ P, thì $L \in$ NP, bởi nếu có một thuật toán thời gian đa thức để quyết định L, thuật toán có thể dễ dàng được chuyển đổi thành một thuật toán xác minh hai-đối số đơn giản bỏ qua mọi chứng chỉ và chấp nhận chính xác các chuỗi đầu vào mà nó xác định nằm trong L. Như vậy, P ⊆ NP.

Ta vẫn chưa biết P = NP hay không, nhưng hầu hết các nhà nghiên cứu đều tin rằng P và NP không cùng lớp. Theo trực giác, lớp P bao gồm các bài toán có thể giải nhanh chóng. Lớp NP bao gồm các bài toán mà một giải pháp có thể được xác minh nhanh chóng. Có lẽ qua kinh nghiệm bạn đã biết rằng việc giải một bài toán từ đầu thường khó khăn hơn việc xác minh một nghiệm đã được trình bày rõ ràng, nhất là khi làm việc dưới các hạn chế về thời gian. Các nhà khoa học máy tính lý thuyết thường tin rằng sự tương tự này mở rộng ra các lớp P và NP, và như vậy tin rằng NP bao gồm các ngôn ngữ không nằm trong P.

Có bằng chứng hấp dẫn hơn rằng P ≠ NP—sự tồn tại của các ngôn ngữ "đầy đủ NP". Ta sẽ nghiên cứu lớp này trong Đoạn 36.3.

Có nhiều câu hỏi căn bản khác vượt quá câu hỏi P ≠ NP vẫn chưa giải quyết. Mặc dù đã có nhiều công sức của nhiều nhà nghiên cứu, thậm chí không ai biết lớp NP có bị đóng dưới phép bù hay không. Nghĩa là, $L \in$ NP có hàm ý $\overline{L} \in$ NP không? Ta có thể định nghĩa ***lớp phức* co-NP** [complexity class co-NP] dưới dạng tập hợp các ngôn ngữ L sao cho $\overline{L} \in$ NP. Câu hỏi NP có bị đóng dưới phép bù hay không có thể được đặt lại là NP = co-NP hay không. Bởi P bị đóng dưới phép bù (Bài tập 36.1-7), nên dẫn đến P ⊆ NP ∩ co-NP. Tuy nhiên, một lần nữa, ta vẫn không biết P = NP ∩ co-NP hay không hoặc có một ngôn ngữ nào đó trong NP ∩ co-NP – P hay không. Hình 36.2 nêu bốn bối cảnh khả dĩ.

Như vậy, sự hiểu biết của chúng ta về mối quan hệ chính xác giữa P và NP hoàn toàn khập kễnh. Tuy vậy, nhờ khảo sát lý thuyết về tính đầy đủ NP, ta sẽ thấy rằng theo quan điểm thực tiễn bất lợi của chúng ta trong việc chứng minh các bài toán là khó trị không phải quá lớn như ta tưởng.



Hình 36.2 Bốn khả năng cho các mối quan hệ giữa các lớp phức. Trong mỗi sơ đồ, một vùng bao một vùng khác nêu rõ một hệ thức tập hợp con riêng. **(a)** P = NP = co-NP. Hầu hết các nhà nghiên cứu đều xem khả năng này là không chắc xảy ra nhất. **(b)** Nếu NP được đóng dưới phép bù, thì NP = co-NP, nhưng không nhất thiết phải là trường hợp P = NP. **(c)** P = NP ∩ co-NP, nhưng NP không bị đóng dưới phép bù. **(d)** NP ≠ co-NP và P ≠ NP ∩ co-NP. Hầu hết các nhà nghiên cứu xem khả năng này là có thể xảy ra nhất.

Bài tập

36.2-1

Xét ngôn ngữ GRAPH-ISOMORPHISM = {<G_1, G_2> : G_1 và G_2 là đồ thị đẳng cấu}. Chứng minh GRAPH-ISOMORPHISM ∈ NP bằng cách mô tả một thuật toán thời gian đa thức để xác minh ngôn ngữ.

36.2-2

Chứng minh nếu G là một đồ thị hai nhánh không hướng có một số lẻ các đỉnh, thì G là phi hamilton.

36.2-3

Chứng tỏ nếu HAM-CYCLE ∈ P, thì bài toán liệt kê các đỉnh của một chu trình hamilton, theo thứ tự, là giải được theo thời gian đa thức.

36.2-4

Chứng minh lớp NP các ngôn ngữ được đóng dưới phép hợp, giao, ghép nối, và sao Kleene. Mô tả bao đóng của NP dưới phép bù.

36.2-5

Chứng tỏ mọi ngôn ngữ trong NP có thể được quyết định bằng một thuật toán chạy trong thời gian $2^{O(n^k)}$ với một hằng k.

36.2-6

Một ***lộ trình hamilton*** trong một đồ thị là một lộ trình đơn giản ghé thăm mọi đỉnh chính xác một lần. Chứng tỏ ngôn ngữ HAM-PATH = {<*G*, *u*, *v*> : có một lộ trình hamilton từ *u* đến *v* trong đồ thị *G*} thuộc về NP.

36.2-7

Chứng tỏ bài toán lộ trình hamilton có thể được giải trong thời gian đa thức trên đồ thị phi chu trình có hướng. Nêu một thuật toán hiệu quả cho bài toán.

36.2-8

Cho ϕ là một công thức bool được kiến tạo từ các biến đầu vào bool $x_1, x_2, \ldots, x_k$, các phép phủ định ($\neg$), các AND ($\wedge$), các OR ($\vee$), và các dấu ngoặc đơn. Công thức ϕ là một ***hằng hiệu*** [tautology] nếu nó đánh giá theo 1 với mọi phép gán 1 và 0 cho các biến đầu vào. Định nghĩa TAUTOLOGY dưới dạng ngôn ngữ của các công thức bool là các hằng hiệu. Chứng tỏ TAUTOLOGY $\in$ co-NP.

36.2-9

Chứng minh P $\subseteq$ co-NP.

36.2-10

Chứng minh nếu NP $\neq$ co-NP, thì P $\neq$ NP.

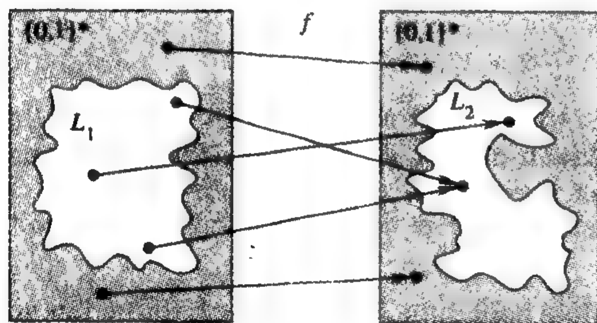
36.2-11

Cho *G* là một đồ thị không hướng, liên thông, có ít nhất G^3 đỉnh, và cho *G* là đồ thị có được bằng cách nối tất cả các cặp đỉnh được nối bởi một lộ trình trong *G* có chiều dài tối đa 3. Chứng minh G^3 là hamilton. (*Mách nước:* Kiến tạo một cây tỏa nhánh cho *G*, và dùng một lập luận quy nạp.)

36.3 Tính đầy đủ NP và khả năng rút gọn

Có lẽ lý do hấp dẫn nhất lý giải tại sao các nhà khoa học máy tính lý thuyết tin rằng P $\neq$ NP là sự tồn tại của lớp các bài toán "đầy đủ NP." Lớp này có tính chất gây kinh ngạc rằng nếu *một* bài toán đầy đủ NP bất kỳ có thể được giải trong thời gian đa thức, thì *mọi* bài toán trong NP đều có một nghiệm thời gian đa thức, nghĩa là, P = NP. Tuy nhiên, bất kể đã bao năm nghiên cứu, chưa có thuật toán thời gian đa thức nào được khám phá với bài toán đầy đủ NP

Ngôn ngữ HAM-CYCLE là một bài toán đầy đủ NP. Nếu có thể quyết định HAM-CYCLE trong thời gian đa thức, thì ta có thể giải mọi bài toán trong NP trong thời gian đa thức. Thực vậy, nếu NP - P sẽ hóa ra không trống rỗng, ta có thể nói chắc rằng HAM-CYCLE ∈ NP − P.



Hình 36.3 Một minh họa về phép rút gọn thời gian đa thức từ một ngôn ngữ L_1 thành một ngôn ngữ L_2 thông qua một hàm rút gọn *f*. Với mọi đầu vào $x \in \{0,1\}^*$, câu hỏi $x \in L_1$ hay không sẽ có cùng đáp án như câu hỏi $f(x) \in L_2$ hay không.

Theo một nghĩa nào đó, các ngôn ngữ đầy đủ NP là những ngôn ngữ "khó nhất" trong NP. Trong đoạn này, ta sẽ nêu cách so sánh "tính khó" tương đối của các ngôn ngữ dùng một khái niệm chính xác tên "khả năng rút gọn thời gian đa thức." Trước tiên, ta chính thức định nghĩa các ngôn ngữ đầy đủ NP, rồi phác họa một phần chứng minh rằng một ngôn ngữ như vậy, có tên CIRCUIT-SAT, là đầy đủ NP. Trong Đoạn 36.5, ta sẽ dùng khái niệm về khả năng rút gọn [reducibility] để chứng tỏ nhiều bài toán khác là đầy đủ NP.

Khả năng rút gọn

Theo trực giác, một bài toán *Q* có thể được rút gọn thành một bài toán *Q'* khác nếu mọi minh dụ của *Q* có thể "dễ dàng được đặt lại" dưới dạng một minh dụ của *Q'*, nghiệm mà một nghiệm cung cấp cho minh dụ của *Q*. Ví dụ, bài toán giải các phương trình tuyến tính trong một trung gian *x* sẽ rút gọn thành bài toán giải các phương trình bậc hai. Cho một minh dụ $ax + b = 0$, ta biến đổi nó thành $0x^2 + ax + b = 0$, có nghiệm cung cấp một nghiệm cho $ax + b = 0$. Như vậy, nếu một bài toán *Q* rút gọn thành một bài toán *Q'* khác, thì theo một nghĩa nào đó, *Q* "không khó giải hơn" so với *Q'*.

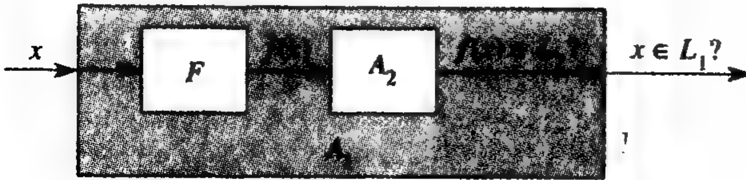
Trở về khuôn khổ ngôn ngữ hình thức của chúng ta cho các bài toán quyết định, ta nói rằng một ngôn ngữ L_1 là ***rút gọn được theo thời gian đa thức*** thành một ngôn ngữ L_2, được viết là $L_1 \leq_P L_2$, nếu ở đó tồn tại một hàm tính toán được theo thời gian đa thức $f : \{0, 1\}^* \rightarrow \{0,1\}^*$ sao cho với

tất cả $x \in \{0,1\}^*$,

$x \in L_1$ nếu và chỉ nếu $f(x) \in L_2$. (36.1)

Ta gọi hàm f là ***hàm rút gọn***, và một thuật toán thời gian đa thức F tính toán f được gọi là một ***thuật toán rút gọn***.

Hình 36.3 minh họa ý tưởng của một phép rút gọn thời gian đa thức từ một ngôn ngữ L_1 thành một ngôn ngữ L_2 khác. Mỗi ngôn ngữ là một tập hợp con $\{0,1\}^*$. Hàm rút gọn f cung cấp một phép ánh xạ thời gian đa thức sao cho nếu $x \in L_1$, thì $f(x) \in L_2$. Hơn nữa, nếu $x \notin L_1$, thì $f(x) \notin L_2$. Như vậy, hàm rút gọn sẽ ánh xạ mọi minh dụ x của bài toán quyết định mà ngôn ngữ L_1 biểu thị theo một minh dụ $f(x)$ của bài toán mà L_2 biểu diễn. Việc cung cấp một đáp án cho câu $f(x) \in L_2$ hay không sẽ trực tiếp cung cấp đáp án cho câu $x \in L_1$ hay không.



Hình 36.4 Phần chứng minh của Bổ đề 36.3. Thuật toán F là một thuật toán rút gọn sẽ tính toán hàm rút gọn f từ L_1 thành L_2 trong thời gian đa thức, và A_2 là một thuật toán thời gian đa thức sẽ quyết định L_2. Được minh họa là một thuật toán A_1 quyết định $x \in L_1$ hay không, bằng cách dùng F để biến đổi mọi đầu vào x thành $f(x)$ rồi dùng A_2 để quyết định $f(x) \in L_2$ hay không.

Các phép rút gọn thời gian đa thức cho ta một công cụ mạnh để chứng minh rằng các ngôn ngữ khác nhau thuộc về P.

Bổ đề 36.3

Nếu $L_1, L_2 \subseteq \{0, 1\}^*$ là các ngôn ngữ sao cho $L_1 \leq_P L_2$, thì $L_2 \in$ P hàm ý $L_1 \in$ P.

Chứng minh Cho A_2 là một thuật toán thời gian đa thức quyết định L_2, và cho F là một thuật toán rút gọn thời gian đa thức sẽ tính toán hàm rút gọn f. Ta sẽ kiến tạo một thuật toán thời gian đa thức A_1 quyết định L_1.

Hình 36.4 minh họa phần kiến tạo của A_1. Với một đầu vào đã cho $x \in \{0, 1\}^*$, thuật toán A_1 sử dụng F để biến đổi x thành $f(x)$, rồi nó sử dụng A_2 để kiểm tra xem $f(x) \in L_2$ hay không. Kết xuất của A_2 là giá trị được cung cấp dưới dạng kết xuất từ A_1.

Tính đúng đắn của A_1 là do điều kiện (36.1). Thuật toán chạy trong thời gian đa thức, bởi cả F lẫn A_2 đều chạy trong thời gian đa thức (xem Bài tập 36.1-6).

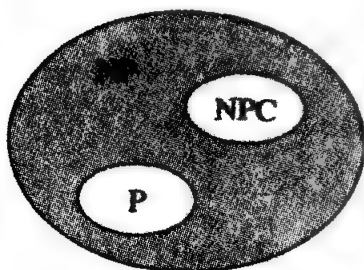
Tính đầy đủ NP

Các phép rút gọn đa thức thời gian cung cấp một biện pháp hình thức để chứng tỏ một bài toán ít nhất cũng khó bằng một bài toán khác, theo một thừa số thời gian đa thức. Nghĩa là, nếu $L_1 \leq_p L_2$, thì L_1 không hơn một thừa số đa thức khó hơn L_2, là lý do tại sao hệ ký hiệu "nhỏ hơn hoặc bằng với" của phép rút gọn lại mang tính gợi nhớ. Giờ đây ta có thể định nghĩa tập hợp các ngôn ngữ đầy đủ NP, là các bài toán khó nhất trong NP.

Một ngôn ngữ $L \subseteq \{0, 1\}^*$ là ***đầy đủ NP*** nếu

1. $L \in$ NP, và

2. $L' \leq_p L$ với mọi $L' \in$ NP.



Hình 36.5 Cách thức mà hầu hết các nhà khoa học máy tính lý thuyết xem các mối quan hệ giữa P, NP, và NPC. Cả P lẫn NPC được chứa hoàn toàn trong NP, và P $\cap$ NPC $\neq \emptyset$.

Nếu một ngôn ngữ L thỏa tính chất 2, nhưng không nhất thiết là tính chất 1, ta nói rằng L là ***khó NP*** [NP-hard]. Ta cũng định nghĩa NPC là lớp các ngôn ngữ đầy đủ NP.

Như định lý dưới đây đã nêu, tính đầy đủ NP là điểm then chốt của việc quyết định P thực tế có bằng với NP hay không.

Định lý 36.4

Nếu một bài toán đầy đủ NP bất kỳ là giải được theo thời gian đa thức, thì P = NP. Nếu một bài toán trong NP bất kỳ không giải được theo thời gian đa thức, thì tất cả các bài toán đầy đủ NP đều không giải được theo thời gian đa thức.

Chứng minh Giả sử $L \in$ P và $L \in$ NPC. Với bất kỳ $L' \in$ NP, ta có $L' \leq_p L$ theo tính chất 2 của phần định nghĩa về tính đầy đủ NP. Như vậy, theo Bổ đề 36.3, ta cũng có $L' \in$ P, chứng minh phát biểu đầu tiên của bổ đề.

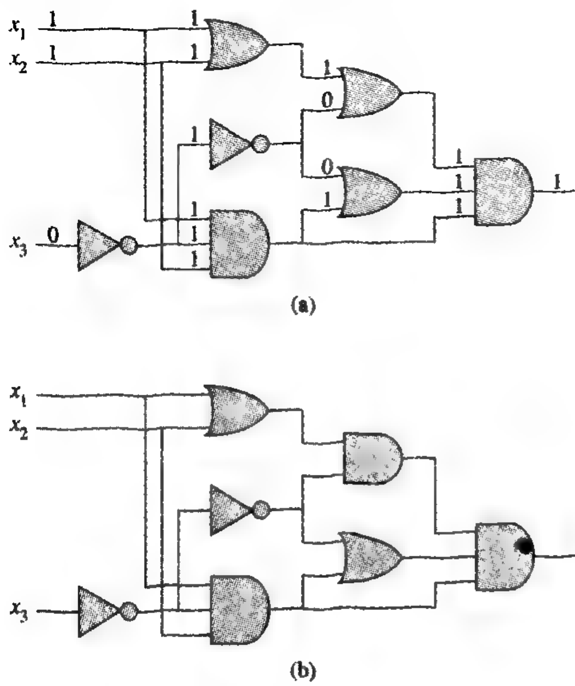
Để chứng minh phát biểu thứ hai, giả sử ở đó tồn tại một $L \in$ NP sao cho $L \notin$ P. Cho $L' \in$ NPC là một ngôn ngữ đầy đủ NP bất kỳ, và vì sự mâu thuẫn, mặc nhận rằng $L' \in$ P. Nhưng như vậy, theo Bổ đề 36.3, ta

có $L \leq_p L'$, và như vậy $L \in$ P.

Chính vì lý do này mà cuộc nghiên cứu vào câu hỏi P ≠ NP tập trung quanh các bài toán đầy đủ NP. Hầu hết các nhà khoa học máy tính lý thuyết đều tin rằng P ≠ NP, dẫn đến các mối quan hệ giữa *P*, NP, và NPC nêu trong Hình 36.5. Nhưng với tất cả những gì chúng ta biết, ai đó có thể nghĩ ra một thuật toán thời gian đa thức cho một bài toán đầy đủ NP, như vậy chứng minh rằng P = NP. Tuy vậy, do chưa có một thuật toán thời gian đa thức nào cho bất kỳ bài toán đầy đủ NP được khám phá, nên một chứng minh cho rằng một bài toán là đầy đủ NP sẽ cung cấp một bằng chứng tuyệt vời cho tính khó trị của nó.

Khả năng thỏa mãn mạch

Ta đã định nghĩa khái niệm của một bài toán đầy đủ NP, nhưng cho đến giờ, ta chưa thực tế chứng minh rằng một bài toán bất kỳ là đầy đủ NP. Một khi ta chứng minh có ít nhất một bài toán đầy đủ NP, ta có thể dùng khả năng rút gọn thời gian đa thức làm một công cụ để chứng minh tính đầy đủ NP của các bài toán khác. Như vậy, giờ đây ta tập trung vào việc chứng minh sự tồn tại của một bài toán đầy đủ NP: bài toán khả năng thỏa mãn mạch.



Hình 36.6 Hai minh dụ của bài toán khả năng thỏa mãn mạch. **(a)** Phép gán $\langle x_1 = 1, x_2 = 1, x_3 = 0\rangle$ cho các đầu vào của mạch này khiến kết xuất của mạch là 1. Do đó mạch là thỏa mãn được. **(b)** Không có phép gán nào cho các đầu vào của mạch này có thể khiến kết xuất của mạch là 1. Do đó mạch không thỏa mãn được.

Đáng tiếc, phần chứng minh chính thức rằng bài toán khả năng thỏa mãn mạch là đầy đủ NP lại yêu cầu chi tiết kỹ thuật vượt quá phạm vi của tài liệu này. Thay vì thế, ta sẽ mô tả không chính thức một chứng minh dựa vào một sự hiểu biết căn bản về các mạch tổ hợp bool. Chất liệu này được xem lại tại đầu Chương 29.

Hình 36.6 nêu hai mạch tổ hợp bool, mỗi mạch có ba đầu vào và một kết xuất đơn lẻ. Một ***phép gán thực*** [truth assignment] cho một mạch tổ hợp bool là một tập hợp các giá trị đầu vào bool. Ta nói rằng một mạch tổ hợp bool kết xuất một là ***thỏa mãn được*** nếu nó có một ***phép gán thỏa*** [satisfying assignment]: một phép gán thực khiến kết xuất của mạch là 1. Ví dụ, mạch trong Hình 36.6(a) có phép gán thỏa $<x_1 = 1, x_2 = 1, x_3 = 0>$, và do đó nó là thỏa mãn được. Không có phép gán giá trị nào cho x_1, x_2, và x_3 khiến mạch trong Hình 36.6(b) tạo ra một kết xuất 1; nó luôn tạo ra 0, và do đó nó không thỏa mãn được.

Bài toán khả năng thỏa mãn mạch là , "Cho một mạch tổ hợp bool bao hàm các cổng AND, OR, và NOT, nó có thỏa mãn được không?" Tuy nhiên, để đặt câu hỏi này một cách chính thức, ta phải đồng ý về một phép mã hóa chuẩn cho các mạch. Ta có thể nghĩ ra một phép mã hóa kiểu đồ thị ánh xạ mọi mạch *C* đã cho vào một chuỗi nhị phân $<C>$ có chiều dài không lớn hơn nhiều so với kích cỡ của chính mạch đó. Do đó, với tư cách là một ngôn ngữ hình thức, ta có thể định nghĩa

CIRCUIT-SAT =

$\{<C>$: C là một mạch tổ hợp bool thỏa mãn được$\}$.

Bài toán khả năng thỏa mãn mạch có tầm quan trọng lớn trong lĩnh vực tối ưu hóa phần cứng nhờ máy tính hỗ trợ. Nếu một mạch luôn tạo ra 0, nó có thể được thay bằng một mạch đơn giản hơn bỏ qua tất cả các cổng logic và cung cấp giá trị 0 không đổi làm kết xuất của nó. Một thuật toán thời gian đa thức cho bài toán sẽ có ứng dụng thực tiễn đáng kể.

Căn cứ vào một mạch *C*, ta có thể gắng xác định xem nó có thỏa mãn được bằng cách đơn giản kiểm tra tất cả các phép gán khả dĩ cho các đầu vào hay không. Đáng tiếc, nếu có *k* đầu vào, ta sẽ có 2^k phép gán khả dĩ. Khi kích cỡ của *C* là đa thức trong *k*, việc kiểm tra từng cái một sẽ dẫn đến một thuật toán thời gian siêu đa thức. Thực vậy, như đã được xác nhận, ta có bằng chứng chắc chắn rằng không có thuật toán thời gian đa thức nào tồn tại để giải bài toán khả năng thỏa mãn mạch bởi khả năng thỏa mãn mạch là đầy đủ NP. Ta tách phần chứng minh sự việc này thành hai phần, dựa trên hai phần của định nghĩa về tính đầy đủ NP.

Bổ đề 36.5

Bài toán khả năng thỏa mãn mạch thuộc về lớp NP.

Chứng minh Ta sẽ cung cấp một thuật toán thời gian đa thức, hai-đầu vào, *A*, có thể xác minh CIRCUIT-SAT. Một trong các đầu vào cho *A* là (một phép mã hóa chuẩn của) một mạch tổ hợp bool *C*. Đầu vào kia là một chứng chỉ tương ứng với một phép gán các giá trị Bool cho các dây dẫn trong *C*.

Thuật toán *A* được kiến tạo như sau. Với mỗi cổng logic trong mạch, nó kiểm tra giá trị mà chứng chỉ cung cấp trên dây dẫn kết xuất được tính toán đúng đắn dưới dạng một hàm của các giá trị trên các dây dẫn đầu vào. Như vậy, nếu kết xuất của nguyên cả mạch là 1, thuật toán kết xuất 1, bởi các giá trị được gán cho các đầu vào của *C* cung cấp một phép gán thỏa. Bằng không, *A* kết xuất 0.

Mỗi khi đầu vào là một mạch thỏa mãn được *C* cho thuật toán *A*, ta có một chứng chỉ có chiều dài là đa thức theo kích cỡ của *C* và khiến *A* kết xuất một 1. Mỗi khi đầu vào là một mạch không thỏa mãn được, không có chứng chỉ nào có thể đánh lừa *A* tin rằng mạch là thỏa mãn được. Thuật toán *A* chạy trong thời gian đa thức: với một thực thi tốt, thời gian tuyến tính là đủ. Như vậy, CIRCUIT-SAT có thể được xác minh trong thời gian đa thức, và CIRCUIT-SAT $\in$ NP.

Phần thứ hai để chứng minh rằng CIRCUIT-SAT là đầy đủ NP đó là chứng tỏ ngôn ngữ là khó NP [NP-hard]. Nghĩa là, ta phải chứng tỏ mọi ngôn ngữ trong NP có thể rút gọn theo thời gian đa thức thành CIRCUIT-SAT. Phần chứng minh thực tế về sự việc này đầy dẫy các chi tiết phức tạp về kỹ thuật, và do đó ta sẽ tạm bằng lòng với một phác thảo của phần chứng minh dựa trên một số hiểu biết về các hoạt động của phần cứng máy tính.

Một chương trình máy tính được lưu trữ trong bộ nhớ máy tính dưới dạng một dãy các chỉ lệnh. Một chỉ lệnh điển hình mã hóa một phép toán sẽ được thực hiện, các địa chỉ của các toán hạng trong bộ nhớ, và một địa chỉ ở đó lưu trữ kết quả. Một vị trí bộ nhớ đặc biệt, có tên ***bộ đếm chương trình***, sẽ theo dõi chỉ lệnh nào sẽ được thi hành kế tiếp. Bộ đếm chương trình tự động được gia số mỗi khi truy nạp một chỉ lệnh, và do đó khiến máy tính thi hành các chỉ lệnh tuần tự. Tuy nhiên, việc thi hành của một chỉ lệnh có thể khiến một giá trị được viết ra bộ đếm chương trình, rồi tiến trình thi hành tuần tự bình thường có thể bị thay đổi, cho phép máy tính lặp vòng và thực hiện các nhánh điều kiện.

Tại một điểm bất kỳ trong khi thi hành một chương trình, nguyên cả trạng thái của phép tính được biểu thị trong bộ nhớ của máy tính. (Ta lấy

bộ nhớ để bao gồm chính chương trình, bộ đếm chương trình, kho lưu trữ làm việc, và bất kỳ trong số nhiều bit trạng thái khác nhau mà một máy tính duy trì để ghi sổ theo dõi.) Ta gọi một trạng thái cụ thể bất kỳ của bộ nhớ máy tính là một ***cấu hình***. Việc thi hành một chỉ lệnh có thể được xem là phép ánh xạ một cấu hình theo một cấu hình khác. Điều quan trọng, phần cứng máy tính hoàn thành phép ánh xạ này có thể được thực thi dưới dạng một mạch tổ hợp bool, mà ta thể hiện bằng M trong phần chứng minh của bổ đề dưới đây.

Bổ đề 36.6

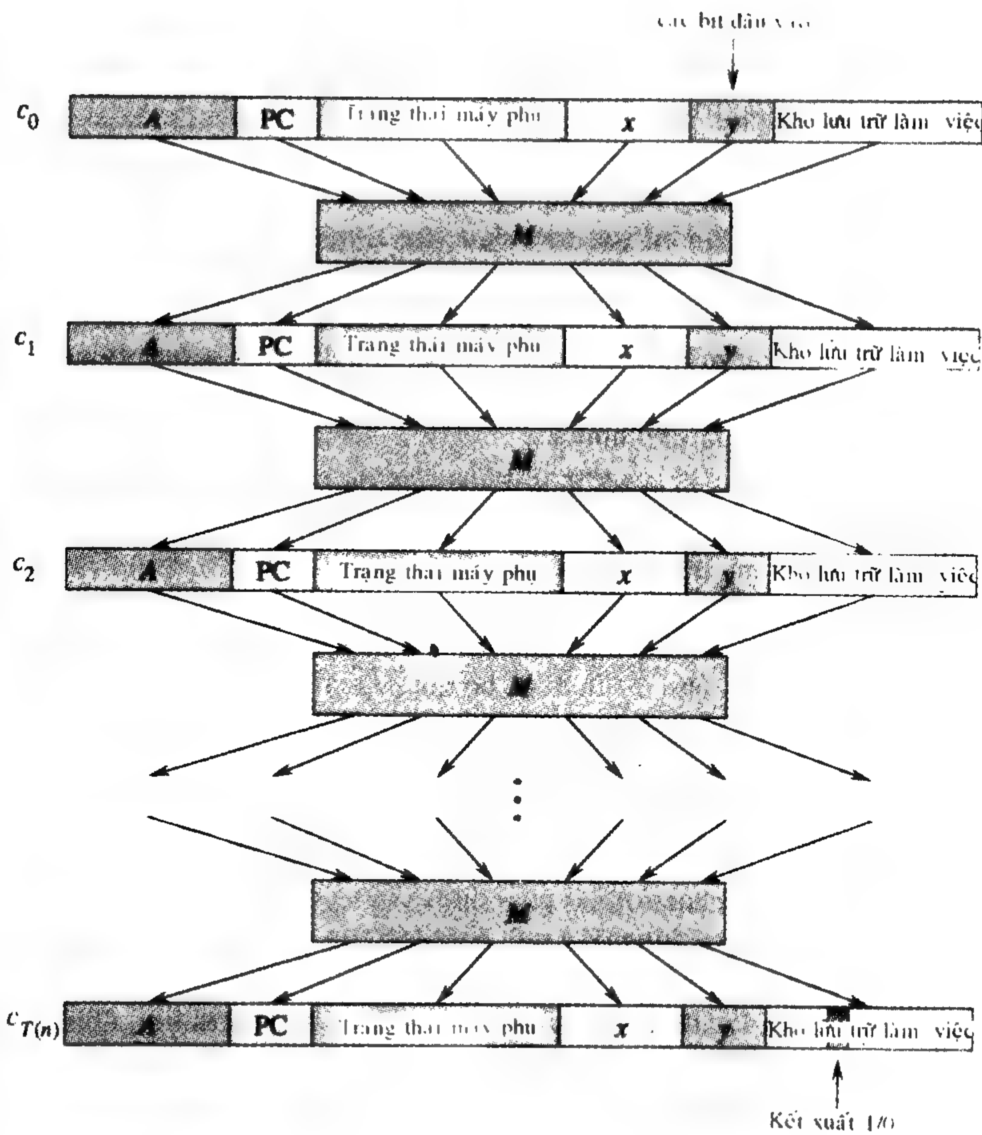
Bài toán khả năng thỏa mãn mạch là khó NP.

Chứng minh Cho L là một ngôn ngữ bất kỳ trong NP. Ta sẽ mô tả một thuật toán thời gian đa thức F tính toán một hàm rút gọn f ánh xạ mọi chuỗi nhị phân x theo một mạch $C = f(x)$ sao cho $x \in L$ nếu và chỉ nếu $C \in$ CIRCUIT-SAT.

Bởi $L \in$ NP, ở đó phải tồn tại một thuật toán A xác minh L trong thời gian đa thức. Thuật toán F mà ta sẽ kiến tạo sẽ dùng thuật toán hai-đầu vào A để tính toán hàm rút gọn f.

Cho $T(n)$ thể hiện thời gian thực hiện ca xấu nhất của thuật toán A trên các chuỗi đầu vào có chiều dài-n, và cho $k \geq 1$ là một hằng sao cho $T(n) = O(n^k)$ và chiều dài của chứng chỉ là $O(n^k)$. (Thời gian thực hiện của A thực tế là một đa thức trong tổng kích cỡ đầu vào, nó bao gồm một chuỗi đầu vào và một chứng chỉ, nhưng bởi chiều dài của chứng chỉ là đa thức trong chiều dài n của chuỗi đầu vào, nên thời gian thực hiện là đa thức trong n.)

Ý tưởng cơ bản của phần chứng minh đó là biểu diễn phép tính của A dưới dạng một dãy các cấu hình. Như đã nêu trong Hình 36.7, mỗi cấu hình có thể được tách nhỏ thành các phần bao gồm chương trình cho A, bộ đếm chương trình và trạng thái máy phụ, đầu vào x, chứng chỉ y, và kho lưu trữ làm việc. Bắt đầu với một cấu hình ban đầu c_0, mỗi cấu hình c_i được ánh xạ theo một cấu hình tiếp theo c_{i+1} bởi mạch tổ hợp M thực thi phần cứng máy tính. Kết xuất của thuật toán A—0 hoặc 1—được ghi ra một vị trí đã chỉ định nào đó trong kho lưu trữ làm việc khi A hoàn tất thi hành, và nếu ta mặc nhận sau đó A treo, giá trị sẽ không bao giờ thay đổi. Như vậy, nếu thuật toán chạy với tối đa $T(n)$ bước, kết xuất xuất hiện dưới dạng một trong các bit trong $c_{T(n)}$.



Hình 36.7 Dãy các cấu hình được tạo bởi một thuật toán A chạy trên một đầu vào x và chứng chỉ y. Mỗi cấu hình biểu diễn trạng thái của máy tính cho một bước của phép tính và, ngoài A, x, và y, bao gồm bộ đếm chương trình (PC), trạng thái máy phụ, và kho lưu trữ làm việc. Ngoại trừ chứng chỉ y, cấu hình ban đầu c_0 không đổi. Mỗi cấu hình được ánh xạ theo cấu hình kế tiếp bởi một mạch tổ hợp bool M. Kết xuất là một bit đặc biệt trong kho lưu trữ làm việc.

Thuật toán rút gọn F kiến tạo một mạch tổ hợp đơn tính toán tất cả các cấu hình được tạo bởi một cấu hình ban đầu đã cho. Ý tưởng đó là dán với nhau $T(n)$ bản sao của mạch M. Kết xuất của mạch thứ i, tạo ra cấu hình c_i, được nạp trực tiếp vào đầu vào của mạch thứ $(i + 1)$. Như vậy, thay vì kết thúc trong một thanh ghi trạng thái, các cấu hình đơn giản thường trú dưới dạng các giá trị trên các dây dẫn nối các bản sao của M.

Hãy nhớ lại nội dung phải thực hiện của thuật toán rút gọn thời gian đa thức F. Căn cứ vào một đầu vào x, nó phải tính toán một mạch $C = f(x)$ thỏa mãn được nếu và chỉ nếu ở đó tồn tại một chứng chỉ y sao cho $A(x, y) = 1$. Khi F có được một đầu vào x, trước tiên nó tính toán $n = |x|$ và kiến tạo một mạch tổ hợp C' bao gồm $T(n)$ bản sao của M. Đầu vào cho C' là một cấu hình ban đầu tương ứng với một phép tính trên $A(x, y)$, và kết xuất là cấu hình $c_{T(n)}$.

Mạch $C = f(x)$ mà F tính toán có được bằng cách sửa đổi chút đỉnh C'. Trước hết, các đầu vào cho C' tương ứng với chương trình của A, bộ đếm chương trình ban đầu, đầu vào x, và trạng thái ban đầu của bộ nhớ được đấu dây trực tiếp với các giá trị đã biết này. Như vậy, các đầu vào duy nhất còn lại cho mạch sẽ tương ứng với chứng chỉ y. Thứ hai, tất cả các kết xuất cho mạch được bỏ qua, ngoại trừ bit một của $c_{T(n)}$ tương ứng với kết xuất của A. Mạch C này, một khi được kiến tạo, sẽ tính toán $C(y) = A(x, y)$ với bất kỳ đầu vào y có chiều dài $O(n^k)$. Khi được cung cấp một chuỗi đầu vào x, thuật toán rút gọn F tính toán một mạch C như vậy và kết xuất nó.

Hai tính chất còn lại phải được chứng minh. Trước hết, ta phải chứng tỏ F tính toán đúng đắn một hàm rút gọn f. Nghĩa là, ta phải chứng tỏ C là thỏa mãn được nếu và chỉ nếu ở đó tồn tại một chứng chỉ y sao cho $A(x, y) = 1$. Thứ hai, ta phải chứng tỏ F chạy trong thời gian đa thức.

Để chứng tỏ F tính toán đúng đắn một hàm rút gọn, ta hãy giả sử ở đó tồn tại một chứng chỉ y có chiều dài $O(n^k)$ sao cho $A(x, y) = 1$. Như vậy, nếu ta áp dụng các bit của y cho các đầu vào của C, kết xuất của C là $C(y) = A(x, y) = 1$. Như vậy, nếu một chứng chỉ tồn tại, thì C là thỏa mãn được. Với hướng kia, giả sử rằng C là thỏa mãn được. Như vậy, ở đó tồn tại một đầu vào y cho C sao cho $C(y) = 1$, từ đó ta kết luận rằng $A(x, y) = 1$. Như vậy, F tính toán đúng đắn một hàm rút gọn.

Để hoàn tất phần chứng minh, ta chỉ cần chứng tỏ F chạy trong thời gian đa thức trong $n = |x|$. Nhận xét đầu tiên đó là số lượng bit cần có để biểu diễn một cấu hình là đa thức trong n. Chương trình cho chính A có kích cỡ không đổi, độc lập với chiều dài đầu vào x của nó. Chiều dài đầu vào x là n, và chiều dài chứng chỉ y là $O(n^k)$. Bởi thuật toán chạy với

tối đa $O(n^k)$ bước, lượng kho lưu trữ làm việc mà A yêu cầu cũng là đa thức trong n. (Ta mặc nhận rằng bộ nhớ này là tiếp cận; Bài tập 36.3-4 yêu cầu bạn mở rộng lập luận cho tình huống ở đó các vị trí được A truy cập nằm rải rác trên một vùng bộ nhớ lớn hơn nhiều và khuôn mẫu tán xạ cụ thể có thể khác nhau cho từng đầu vào x.)

Mạch tổ hợp M thực thi phần cứng máy tính có kích cỡ đa thức trong chiều dài của một cấu hình, là đa thức trong $O(n^k)$ và do đó là đa thức trong n. (Hầu hết hệ mạch này thực thi logic của hệ thống bộ nhớ.) Mạch C bao gồm tối đa $t = O(n^k)$ bản sao của M, và do đó nó có kích cỡ đa thức trong n. Phần kiến tạo của C từ x có thể được hoàn thành trong thời gian đa thức bằng thuật toán rút gọn F, bởi mỗi bước của phần kiến tạo chiếm thời gian đa thức.

Do đó, ngôn ngữ CIRCUIT-SAT ít nhất cũng khó bằng bất kỳ ngôn ngữ nào trong NP, và bởi nó thuộc về NP, nên nó đầy đủ NP.

Định lý 36.7

Bài toán khả năng thỏa mãn mạch là đầy đủ NP.

Chứng minh Tức thời từ các Bổ đề 36.5 và 36.6 và phần định nghĩa của tính đầy đủ NP.

Bài tập

36.3-1

Chứng tỏ hệ thức $\leq_P$ là một hệ thức bắc cầu trên các ngôn ngữ. Nghĩa là, chứng tỏ nếu $L_1 \leq_P L_2$ và $L_2 \leq_P L_3$, thì $L_1 \leq_P L_3$.

36.3-2

Chứng minh $L \leq_P \overline{L}$ nếu và chỉ nếu $\overline{L} \leq_P L$.

36.3-3

Chứng tỏ có thể dùng một phép gán thỏa làm một chứng chỉ trong một chứng minh khác của Bổ đề 36.5. Chứng chỉ nào giúp cho việc chứng minh trở nên dễ hơn?

36.3-4

Phần chứng minh của Bổ đề 36.6 mặc nhận kho lưu trữ làm việc cho thuật toán A choán một vùng tiếp cận có kích cỡ đa thức. Giả thiết này được khai thác ở đâu trong phần chứng minh? Chứng tỏ giả thiết này không liên quan đến sự mất mát tính tổng quát.

36.3-5

Một ngôn ngữ L là ***đầy đủ*** cho một lớp ngôn ngữ C đối với các phép

rút gọn theo thời gian đa thức nếu $L \in C$ và $L' \leq_p L$ với tất cả $L' \in C$. Chứng tỏ $\emptyset$ và $\{0,1\}^*$ là những ngôn ngữ duy nhất trong P không đầy đủ cho P đối với các phép rút gọn thời gian đa thức.

36.3-6

Chứng tỏ L là đầy đủ cho NP nếu và chỉ nếu $\overline{L}$ là đầy đủ cho co-NP.

36.3-7

Thuật toán rút gọn F trong phần chứng minh của Bổ đề 36.6 kiến tạo mạch $C = f(x)$ dựa trên kiến thức về x, A, và k. Giáo sư Sartre nhận thấy chuỗi x là đầu vào cho F, nhưng F chỉ biết sự tồn tại của A và k (bởi ngôn ngữ L thuộc về NP), chứ không phải các giá trị thực tế của chúng. Như vậy, giáo sư kết luận rằng F không thể kiến tạo mạch C và ngôn ngữ CIRCUIT-SAT không nhất thiết là khó NP. Giải thích chỗ sai lầm trong biện luận của giáo sư.

36.4 Các chứng minh về tính đầy đủ NP

Tính đầy đủ NP của bài toán khả năng thỏa mãn mạch dựa vào một chứng minh trực tiếp rằng $L \leq_p$ CIRCUIT-SAT với mọi ngôn ngữ $L \in$ NP. Trong đoạn này, ta sẽ nêu cách chứng minh rằng các ngôn ngữ là đầy đủ NP mà không trực tiếp rút gọn *mọi* ngôn ngữ trong NP thành ngôn ngữ đã cho. Ta sẽ minh họa phương pháp luận này bằng cách chứng minh rằng các bài toán khả năng thỏa mãn công thức khác nhau là đầy đủ NP. Đoạn 36.5 cung cấp thêm nhiều ví dụ về phương pháp luận.

Bổ đề dưới đây là cơ sở cho phương pháp của chúng ta để chứng tỏ một ngôn ngữ là đầy đủ NP.

Bổ đề 36.8

Nếu L là một ngôn ngữ sao cho $L' \leq_p L$ với vài $L' \in$ NPC, thì L là khó NP. Hơn nữa, nếu $L \in$ NP, thì $L \in$ NPC.

Chứng minh Bởi L' đầy đủ NP, với tất cả $L'' \in$ NP, nên ta có $L'' \leq_p L'$. Theo giả thiết, $L' \leq_p L$, và như vậy theo tính bắc cầu (Bài tập 36.3-1), ta có $L'' \leq_p L$, chứng tỏ L là khó NP. Nếu $L \in$ NP, ta cũng có $L \in$ NPC.

Nói cách khác, bằng cách rút gọn một ngôn ngữ đầy đủ NP đã biết L' thành L, ta mặc định rút gọn mọi ngôn ngữ trong NP thành L. Như vậy, Bổ đề 36.8 cho ta một phương pháp để chứng minh rằng một ngôn ngữ L là đầy đủ NP:

1. Chứng minh $L \in$ NP.

2. Lựa một ngôn ngữ đầy đủ NP đã biết L'.

3. Mô tả một thuật toán tính toán một hàm f ánh xạ mọi minh dụ của L' thành một minh dụ của L.

4. Chứng minh hàm f thỏa $x \in L'$ nếu và chỉ nếu $f(x) \in L$ với tất cả $x \in \{0, 1\}^*$.

5. Chứng minh thuật toán tính toán f chạy trong thời gian đa thức.

Phương pháp luận này rút gọn từ một ngôn ngữ đơn lẻ đầy đủ NP đã biết thường tỏ ra đơn giản hơn nhiều so với tiến trình phức tạp hơn cung cấp các phép rút gọn từ mọi ngôn ngữ trong NP. Việc chứng minh CIRCUIT-SAT ∈ NPC đã đưa ta một "chân vào cửa." Giờ đây biết được bài toán khả năng thỏa mãn mạch là đầy đủ NP sẽ cho phép ta chứng minh dễ dàng hơn nhiều rằng các bài toán khác là đầy đủ NP. Hơn nữa, khi ta phát triển một catolô các bài toán đầy đủ NP đã biết, việc áp dụng phương pháp luận sẽ trở thành dễ dàng hơn nhiều.

Khả năng thỏa mãn công thức

Ta minh họa phương pháp luận rút gọn bằng cách đưa ra phần chứng minh tính đầy đủ NP cho bài toán xác định xem một công thức Bool, chứ không phải một mạch, có thỏa mãn được hay không. Bài toán này có niềm vinh dự lịch sử là bài toán đầu tiên được nêu là đầy đủ NP.

Ta trình bày bài toán ***khả năng thỏa mãn (công thức)*** theo dạng ngôn ngữ SAT như sau. Một minh dụ của SAT là một công thức Bool ϕ bao gồm

1. Các biến Bool: $x_1, x_2, \ldots$;

2. Các toán tử lôgic: mọi hàm bool có một hoặc hai đầu vào và một kết xuất, như $\wedge$ (AND), $\vee$ (OR), $\neg$ (NOT), $\rightarrow$ (phép tất suy), $\leftrightarrow$ (nếu và chỉ nếu); và

3. Các dấu ngoặc đơn.

Như trong các mạch tổ hợp bool, một ***phép gán thực*** cho một công thức Bool ϕ là một tập hợp các giá trị với các biến của ϕ, và một ***phép gán thỏa*** là một phép gán thực khiến nó đánh giá là 1. Một công thức có một phép gán thỏa là một công thức ***thỏa mãn được***. Bài toán khả năng thỏa mãn yêu cầu một công thức Bool đã cho có thỏa mãn được hay không; trong dạng ngôn ngữ hình thức,

SAT = {<ϕ> : ϕ là một công thức Bool thỏa mãn được} .

Để lấy ví dụ, công thức

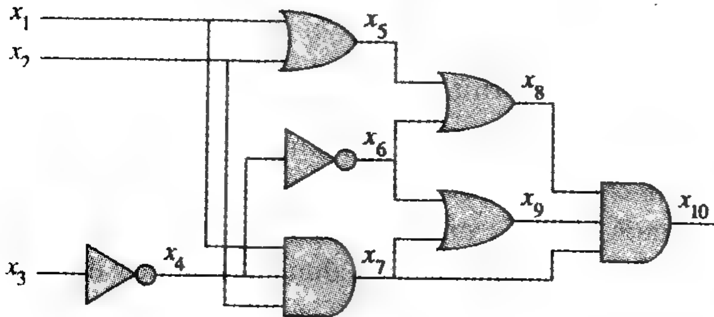
$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

có phép gán thỏa $\langle x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1\rangle$, bởi

$$\begin{aligned}\phi &= ((0 \to 0) \vee \neg((\neg 0 \leftrightarrow 1) \vee 1)) \wedge \neg 0 \qquad (36..2)\\ &= (1 \vee \neg(1 \vee 1)) \wedge 1\\ &= (1 \vee 0) \wedge 1\\ &= 1,\end{aligned}$$

và như vậy công thức ϕ *này* thuộc về SAT.

Thuật toán đơn sơ để xác định một công thức Bool tùy ý có thỏa mãn được hay không sẽ không chạy trong thời gian đa thức. Có 2^n phép gán khả dĩ trong một công thức ϕ với n biến. Nếu chiều dài của $\langle\phi\rangle$ là đa thức trong n, thì việc kiểm tra mọi phép gán sẽ yêu cầu thời gian siêu đa thức. Như định lý dưới đây đã nêu, một thuật toán thời gian đa thức ắt không tồn tại.



Hình 36.8 Rút gọn khả năng thỏa mãn mạch thành khả năng thỏa mãn công thức. Công thức được tạo bởi thuật toán rút gọn có một biến cho mỗi dây dẫn trong mạch.

Định lý 36.9

Khả năng thỏa mãn của các công thức bool là đầy đủ NP.

Chứng minh Trước tiên ta sẽ chứng minh rằng SAT $\in$ NP. Sau đó, ta sẽ chứng tỏ CIRCUIT-SAT $\leq_P$ SAT; theo Bổ đề 36.8, điều này sẽ chứng minh định lý.

Để chứng tỏ SAT thuộc về NP, ta chứng tỏ một chứng chỉ bao gồm một phép gán thỏa dành cho một công thức đầu vào ϕ có thể được xác minh trong thời gian đa thức. Thuật toán xác minh đơn giản thay mỗi biến trong công thức bằng giá trị tương ứng của nó rồi đánh giá biểu thức, tương tự như đã làm trong phương trình (36.2) trên đây. Công việc này dễ thực hiện được trong thời gian đa thức. Nếu biểu thức đánh giá là 1, công thức là thỏa mãn được. Như vậy, điều kiện đầu tiên của Bổ đề 36.8 về tính đầy đủ NP sẽ đứng vững.

Để chứng minh SAT là khó NP, ta chứng tỏ CIRCUIT-SAT $\leq_P$ SAT.

Nói cách khác, mọi minh dụ của khả năng thỏa mãn mạch đều có thể được rút gọn trong thời gian đa thức thành một minh dụ của khả năng thỏa mãn công thức. Phương pháp quy nạp có thể được dùng để diễn tả các mạch tổ hợp bool dưới dạng một công thức bool. Ta đơn giản xét cổng tạo ra mạch kết xuất và theo quy nạp diễn tả mỗi trong số các đầu vào của cổng dưới dạng các công thức. Như vậy, công thức của mạch có được bằng cách viết một biểu thức áp dụng chức năng của cổng cho các công thức của đầu vào của nó.

Đáng tiếc, phương pháp đơn giản này không hình thành một phép rút gọn thời gian đa thức. Các công thức con dùng chung có thể khiến kích cỡ của công thức phát sinh tăng trưởng theo hàm mũ (xem Bài tập 36.4-1). Như vậy, thuật toán rút gọn phải hơi thông minh hơn.

Hình 36.8 minh họa ý tưởng cơ bản của phép rút họn từ CIRCUIT-SAT thành SAT trên mạch trong Hình 36.6(a). Với mỗi dây dẫn x_i trong mạch C, công thức ϕ có một biến x_i. Phép toán đúng của một cổng giờ đây có thể được diễn tả dưới dạng một công thức bao hàm các biến của các dây dẫn liên thuộc của nó. Ví dụ, phép toán của cổng AND kết xuất là $x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)$.

Công thức ϕ được tạo bởi thuật toán rút gọn là AND của biến mạch-kết xuất với phép hội của các mệnh đề mô tả phép toán của mỗi cổng. Với mạch trong hình, công thức là

$$\begin{aligned}
\phi \quad = \quad x_{10} \quad & \wedge \quad (x_4 \leftrightarrow \neg x_3) \\
& \wedge \quad (x_5 \leftrightarrow (x_1 \vee x_2)) \\
& \wedge \quad (x_6 \leftrightarrow \neg x_4) \\
& \wedge \quad (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4)) \\
& \wedge \quad (x_8 \leftrightarrow (x_5 \vee x_6)) \\
& \wedge \quad (x_9 \leftrightarrow (x_6 \vee x_7)) \\
& \wedge \quad (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)) .
\end{aligned}$$

Cho một mạch C, ta có thể dễ dàng tạo ra một công thức ϕ như vậy trong thời gian đa thức.

Tại sao mạch ϕ thỏa mãn được chính xác khi công thức ϕ thỏa mãn được? Nếu C có một phép gán thỏa, mỗi dây dẫn của mạch có một giá trị được định nghĩa kỹ, và kết xuất của mạch là 1. Do đó, phép gán của các giá trị dây dẫn cho các biến trong ϕ khiến mỗi mệnh đề của ϕ đánh giá là 1, và như vậy phép hội của tất cả sẽ đánh giá là 1. Ngược lại, nếu có một phép gán khiến ϕ đánh giá là 1, mạch C là thỏa mãn được theo một lập luận tương tự. Như vậy, ta đã chứng tỏ CIRCUIT-SAT $\leq_P$ SAT, hoàn tất phần chứng minh.

Khả năng thỏa mãn 3-CNF

Nhiều bài toán có thể được chứng minh đầy đủ NP bằng phép rut gọn từ khả năng thỏa mãn công thức. Tuy vậy, thuật toán rút gọn phải điều quản mọi công thức đầu vào, và điều này có thể dẫn đến một lượng khổng lồ các trường hợp phải được xem xét. Do đó, tốt nhất ta nên rút gọn từ một ngôn ngữ hạn chế của các công thức Bool, để giảm bớt số lượng trường hợp cần được xem xét. Tất nhiên, ta không được hạn chế ngôn ngữ nhiều đến nỗi nó trở thành giải được theo thời gian đa thức. Một ngôn ngữ tiện dụng đó là khả năng thỏa mãn 3-CNF, hoặc 3-CNF-SAT.

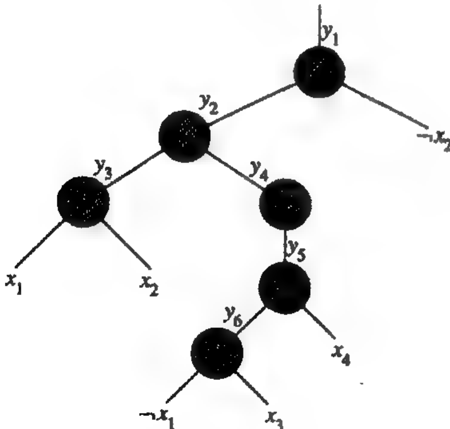
Ta định nghĩa khả năng thỏa mãn 3-CNF bằng các thuật ngữ sau đây. Một *trực kiện* [literal] trong một công thức Bool là một trường hợp xuất hiện của một biến hoặc sự phủ định của nó. Một công thức Bool có ***dạng chuẩn tắc hội*** [conjunctive normal form], hoặc ***CNF***, nếu nó được diễn tả dưới dạng một AND của các mệnh đề, mỗi mệnh đề là OR của một hoặc nhiều trực kiện. Một công thức Bool có ***dạng chuẩn tắc hội-3*** [3-conjunctive normal form], hoặc ***3-CNF***, nếu mỗi mệnh đề có chính xác ba trực kiện riêng biệt.

Ví dụ, công thức Bool

$$(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

thuộc 3-CNF. Mệnh đề đầu tiên trong số ba mệnh đề của nó là $(x_1 \vee \neg x_1 \vee \neg x_2)$, chứa ba trực kiện x_1, $\neg x_1$, và $\neg x_2$.

Trong 3-CNF-SAT, ta được hỏi một công thức Bool ϕ đã cho thuộc 3-CNF có phải thỏa mãn được không. Định lý dưới đây chứng tỏ một thuật toán thời gian đa thức có thể xác định khả năng thỏa mãn của các công thức Bool hầu như không tồn tại, thậm chí khi chúng được diễn tả theo dạng chuẩn tắc đơn giản này.



Hình 36.9 Cây tương ứng với công thức $\phi = ((x_1 \to x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$.

Định lý 36.10

Khả năng thỏa mãn của các công thức bool thuộc 3-CNF là đầy đủ NP.

Chứng minh Lập luận mà ta đã dùng trong phần chứng minh của Định lý 36.9 để chứng tỏ SAT $\in$ NP cũng được áp dụng ở đây để chứng tỏ 3-CNF-SAT $\in$ NP. Như vậy, ta chỉ cần chứng tỏ 3-CNF-SAT là khó NP. Ta chứng minh điều này bằng cách chứng tỏ SAT $\leq_p$ 3-CNF-SAT, từ đó phần chứng minh sẽ nảy sinh theo Bổ đề 36.8.

Thuật toán rút gọn có thể được tách nhỏ thành ba bước cơ bản. Mỗi bước từ từ biến đổi công thức đầu vào ϕ sát hơn với 3-CNF mong muốn.

Bước đầu tiên tương tự như bước đã được dùng để chứng minh CIRCUIT-SAT $\leq_p$ SAT trong Định lý 36.9. Trước tiên, ta kiến tạo một cây "phân ngữ" nhị phân cho công thức đầu vào ϕ, có các trực kiện làm các lá và các toán tử lôgic làm các mắt trong. Hình 36.9 nêu một cây phân ngữ như vậy cho công thức

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2 \,. \qquad (36.3)$$

Nếu công thức đầu vào chứa một mệnh đề như OR của một vài trực kiện, tính kết hợp có thể được dùng để ngoặc đơn biểu thức một cách đầy đủ sao cho mọi mắt trong trong cây kết quả đều có 1 hoặc 2 con. Giờ đây, cây phân ngữ nhị phân có thể được xem là một mạch để tính toán hàm.

y_1	y_2	x_2	$(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	1

Hình 36.10 Bảng chân trị của mệnh đề $(y_1 \leftrightarrow (y_2 \vee \neg x_2))$.

Bắt chước phép rút gọn trong phần chứng minh của Định lý 36.9, ta giới thiệu một biến y_i với kết xuất của mỗi mắt trong. Như vậy, ta viết lại công thức ϕ ban đầu dưới dạng AND của biến gốc và một phép hội với các mệnh đề mô tả phép toán của mỗi mắt. Với công thức (36.3), biểu thức kết quả là

$$\begin{aligned}\phi' &= y_1 \wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \\ &\wedge (y_2 \leftrightarrow (y_3 \vee y_4)) \\ &\wedge (y_3 \leftrightarrow (x_1 - x_2)) \\ &\wedge (y_4 \leftrightarrow \neg y_5) \\ &\wedge (y_5 \leftrightarrow (y_6 \vee x_4)) \\ &\wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3)).\end{aligned}$$

Nhận thấy công thức ϕ' có được như vậy chính là một phép hội của các mệnh đề ϕ'_i, mỗi mệnh đề có tối đa 3 trực kiện. Yêu cầu bổ sung duy nhất đó là mỗi mệnh đề là một OR của các trực kiện.

Bước thứ hai của phép rút gọn chuyển đổi mỗi mệnh đề ϕ'_i thành dạng chuẩn tắc hội. Ta kiến tạo một bảng chân trị cho ϕ'_i bằng cách đánh giá tất cả các phép gán khả dĩ cho các biến của nó. Mỗi hàng của bảng chân trị bao gồm một phép gán khả dĩ cho các biến của mệnh đề, cùng với giá trị của mệnh đề dưới phép gán đó. Dùng các khoản nhập bảng chân trị đánh giá theo 0, ta xây dựng một công thức thuộc ***dạng chuẩn tắc tuyển*** [disjunctive normal form] (hoặc ***DNF***)—một OR của các AND—tương đương với $\neg\phi'_i$. Sau đó, ta chuyển đổi công thức này thành một công thức CNF ϕ''_i nhờ dùng luật DeMorgan (5.2) để thực thi tất cả các trực kiện và thay đổi các OR thành các AND và các AND thành các OR.

Ví dụ của chúng ta sẽ chuyển đổi mệnh đề $\phi'_i = (y_1 \leftrightarrow (y_2 \wedge \neg x_2))$ thành CNF như sau. Bảng chân trị của ϕ'_1 được nêu trong Hình 36.10. Công thức DNF tương đương với $\neg\phi'_i$ là

$$(y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2).$$

Áp dụng luật DeMorgan, ta có công thức CNF

$$\begin{aligned}\phi''_1 = &(\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \\ &\wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2),\end{aligned}$$

tương đương với mệnh đề ban đầu ϕ'_1.

Mỗi mệnh đề ϕ'_i của công thức ϕ' giờ đây đã được chuyển đổi thành một công thức CNF ϕ''_i, và như vậy ϕ' tương đương với công thức CNF ϕ'' bao gồm phép hội của ϕ''_i. Hơn nữa, mỗi mệnh đề của ϕ'' có tối đa 3 trực kiện.

Bước thứ ba và cuối cùng của phép rút gọn biến đổi công thức hơn nữa sao cho mỗi mệnh đề có *chính xác* 3 trực kiện riêng biệt. Công thức 3-CNF cuối cùng ϕ''' được kiến tạo từ các mệnh đề của công thức CNF ϕ''. Nó cũng sử dụng hai biến phụ mà ta gọi là p và q. Với mỗi mệnh đề C_i của ϕ'', ta gộp các mệnh đề sau đây trong ϕ''':

- Nếu C_i có 3 trực kiện riêng biệt, thì đơn giản gộp C_i dưới dạng một mệnh đề của ϕ'''.

- Nếu C_i có 2 trực kiện riêng biệt, nghĩa là, nếu $C_i = (l_1 \vee l_2)$, ở đó l_1 và l_2 là các trực kiện, thì gộp $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$ dưới dạng các mệnh đề của $f(\phi)$. Các trực kiện p và $\neg p$ đơn thuần đáp ứng yêu cầu cú pháp là có chính xác 3 trực kiện riêng biệt mỗi mệnh đề: $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$ tương đương với $(l_1 \vee l_2)$ dẫu $p = 0$ hoặc $p = 1$.

- Nếu C_i chỉ có 1 trực kiện riêng biệt l, thì gộp $(l \vee p \vee q) \wedge (l \vee p \vee \neg q) \wedge (l \vee \neg p \vee q) \wedge (l \vee \neg p \vee \neg q)$ dưới dạng các mệnh đề của ϕ'''. Lưu ý, mọi xác lập của p và q khiến phép hội của bốn mệnh đề này đánh giá là l.

Ta có thể thấy công thức 3-CNF ϕ''' là thỏa mãn được nếu và chỉ nếu ϕ là thỏa mãn được bằng cách thẩm tra mỗi trong số ba bước. Cũng như phép rút gọn từ CIRCUIT-SAT thành SAT, phần kiến tạo của ϕ' từ ϕ trong bước đầu tiên sẽ bảo toàn khả năng thỏa mãn. Bước thứ hai tạo ra một công thức CNF ϕ'' mà xét theo đại số là tương đương với ϕ'. Bước thứ ba tạo ra một công thức 3-CNF ϕ''' thực sự tương đương với ϕ'', bởi mọi phép gán cho các biến p và q đều tạo ra một công thức mà theo đại số tương đương với ϕ''.

Ta cũng phải chứng tỏ phép rút gọn có thể được tính toán trong thời gian đa thức. Việc kiến tạo ϕ' từ ϕ có thể đưa ra tối đa 1 biến và 1 mệnh đề cho mỗi toán tử lôgic [connective] trong ϕ. Việc kiến tạo ϕ'' từ ϕ' có thể đưa ra tối đa 8 mệnh đề vào ϕ'' cho mỗi mệnh đề từ ϕ', bởi mỗi mệnh đề của ϕ' có tối đa 3 biến, và bảng chân trị của mỗi mệnh đề có tối đa $2^3 = 8$ hàng. Phần kiến tạo của ϕ''' từ ϕ'' đưa ra tối đa 4 mệnh đề vào ϕ''' với mỗi mệnh đề của ϕ''. Như vậy, kích cỡ của công thức kết quả ϕ''' là đa thức theo chiều dài của công thức ban đầu. Từng phần kiến tạo có thể dễ dàng hoàn thành trong thời gian đa thức.

Bài tập

36.4-1

Xét phép rút gọn đơn giản (thời gian phi đa thức) trong phần chứng minh của Định lý 36.9. Mô tả một mạch có kích cỡ n mà, khi được chuyển đổi thành một công thức bằng phương pháp này, sẽ cho ra một công thức có kích cỡ là hàm mũ trong n.

36.4-2

Nêu công thức 3-CNF là kết quả khi ta dùng phương pháp của Định lý 36.10 trên công thức (36.3).

36.4-3

Giáo sư Jagger đề xuất chứng tỏ SAT $\leq_p$ 3-CNF-SAT bằng cách chỉ dùng kỹ thuật bảng chân trị trong phần chứng minh của Định lý 36.10, và không dùng các bước khác. Nghĩa là, giáo sư đề xuất lấy công thức Bool ϕ, hình thành một bảng chân trị với các biến của nó, suy ra từ bảng chân trị một công thức thuộc 3-DNF tương đương với $\neg\phi$, rồi phủ định và áp dụng luật DeMorgan để tạo ra một công thức 3-CNF tương đương với ϕ. Chứng tỏ chiến lược này không cho ra phép rút gọn thời gian đa thức.

36.4-4

Chứng tỏ bài toán xác định một công thức Bool có phải là một hằng hiệu hay không là đầy đủ [complete] với co-NP. (*Mách nước:* Xem Bài tập 36.3-6.)

36.4-5

Chứng tỏ bài toán xác định khả năng thỏa mãn của các công thức Bool theo dạng chuẩn tắc tuyển là giải được theo thời gian đa thức.

36.4-6

Giả sử rằng có người cho bạn một thuật toán thời gian đa thức để quyết định khả năng thỏa mãn công thức. Mô tả cách sử dụng thuật toán này để tìm ra các phép gán thỏa trong thời gian đa thức.

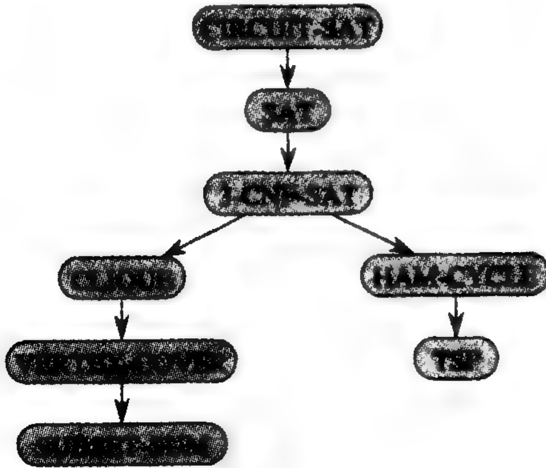
36.4-7

Cho 2-CNF-SAT là tập hợp các công thức Bool thỏa mãn được trong CNF với chính xác 2 trực kiện cho mỗi mệnh đề. Chứng tỏ 2-CNF-SAT $\in$ P. Tạo thuật toán của bạn càng hiệu quả càng tốt. (*Mách nước:* Nhận thấy $x \vee y$ tương đương với $\neg x \rightarrow y$. Rút gọn 2-CNF-SAT thành một bài toán trên một đồ thị có hướng giải được một cách hiệu quả.)

36.5 Các bài toán đầy đủ NP

Các bài toán đầy đủ NP nảy sinh trong các lĩnh vực đa dạng: logic Bool, đồ thị, số học, thiết kế mạng, các tập hợp và các phân hoạch, lưu trữ và truy lục, kiểm tra trình tự và lên lịch, lập trình toán học, đại số học và lý thuyết số, trò chơi và câu đố, otomat và lý thuyết ngôn ngữ, tối ưu hóa chương trình, và vân vân. Trong đoạn này, ta sẽ dùng phương pháp luận rút gọn để cung cấp các chứng minh tính đầy đủ NP cho nhiều bài toán đa dạng được rút từ lý thuyết đồ thị và ấn định tiến trình phân hoạch.

Hình 36.11 nêu khái quát cấu trúc của các chứng minh tính đầy đủ NP trong đoạn này và Đoạn 36.4. Mỗi ngôn ngữ trong hình được chứng minh là đầy đủ NP bằng phép rut gọn từ ngôn ngữ trỏ đến nó. Tại gốc là CIRCUIT-SAT, mà ta đã chứng minh đầy đủ NP trong Định lý 36.7.



Hình 36.11 Cấu trúc của các chứng minh tính đầy đủ NP trong các Đoạn 36.4 và 36.5. Tất cả các chứng minh chung cuộc đều xuất xứ từ CIRCUIT-SAT, theo phép rút gọn.

36.5.1 Bài toán bọn

Một ***bọn*** [clique] trong một đồ thị không hướng $G = (V, E)$ là một tập hợp con $V' \subseteq V$ các đỉnh, mỗi cặp đỉnh được liên thông bằng một cạnh trong E. Nói cách khác, một bọn là một đồ thị con đầy đủ của G. Kích cỡ của một bọn là số lượng các đỉnh mà nó chứa. ***Bài toán bọn*** là bài toán tối ưu hóa để tìm một bọn có kích cỡ cực đại trong một đồ thị. Với tư cách là một bài toán quyết định, ta đơn giản hỏi một bọn có một kích cỡ đã cho k có tồn tại trong đồ thị hay không. Định nghĩa chính thức là

CLIQUE = {$<G, k>$: G là một đồ thị có một bọn có kích cỡ k} .

Một thuật toán đơn sơ để xác định một đồ thị $G = (V, E)$ có $|V|$ đỉnh có một bọn có kích cỡ k hay không đó là liệt kê tất cả k tập hợp con của V, và kiểm tra mỗi tập hợp để xem nó có hình thành một bọn hay không. Thời gian thực hiện của thuật toán này là $\Omega(k^2 \binom{|V|}{k})$, là đa thức nếu k là một hằng. Tuy nhiên, nói chung, k có thể tỷ lệ với $|V|$, trong trường hợp đó thuật toán chạy trong thời gian siêu đa thức. Như ta có thể nghi ngờ, một thuật toán hiệu quả cho bài toán bọn ắt không tồn tại.

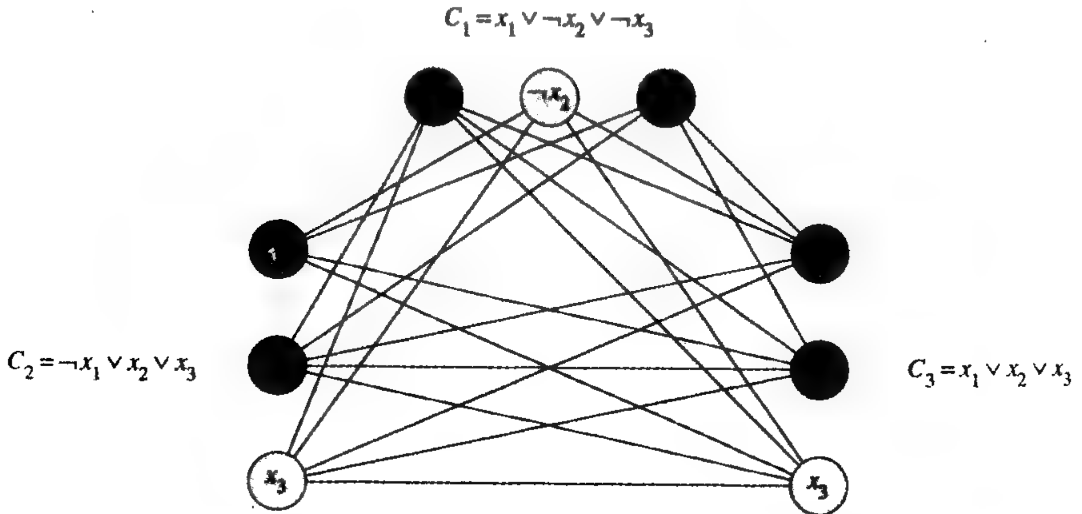
Định lý 36.11

Bài toán bọn là đầy đủ NP.

Chứng minh Để chứng tỏ CLIQUE $\in$ NP, với một đồ thị đã cho $G =$

(V, E), ta dùng tập hợp $V' \subseteq V$ các đỉnh trong bọn dưới dạng một chứng chỉ cho G. Tiến trình kiểm tra V' có phải là một bọn hay không có thể được hoàn thành trong thời gian đa thức bằng cách kiểm tra, với mọi cặp $u, v \in V'$, cạnh (u, v) có thuộc về E hay không.

Kế đó, ta chứng tỏ bài toán bọn là khó NP bằng cách chứng minh 3CNF-SAT $\leq_p$ CLIQUE. Việc mà ta phải chứng minh được kết quả này có phần gây ngạc nhiên, bởi xét bề ngoài các công thức lôgic dường như ít chẳng dính dáng gì mấy đến các đồ thị.



Hình 36.12 Đồ thị G phái sinh từ công thức 3-CNF $\phi = C_1 \wedge C_2 \wedge C_3$, ở đó $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$, $C_2 = (\neg x_1 \vee x_2 \vee x_3)$, và $C_3 = (x_1 \vee x_2 \vee x_3)$, trong khi rút gọn 3-CNF-SAT thành CLIQUE. Một phép gán thỏa của công thức là $\langle x_1 = 0, x_2 = 0, x_3 = 1\rangle$. Phép gán thỏa này thỏa mãn C_1 bằng $\neg x_2$, và nó thỏa mãn C_2 và C_3 bằng x_3, tương ứng với bọn có các đỉnh tô bóng nhạt.

Thuật toán rút gọn bắt đầu bằng một minh dụ của 3-CNF-SAT. Cho $\phi = C_1 \wedge C_2 \wedge \ldots \wedge C_k$ là một công thức Bool thuộc 3-CNF với k mệnh đề. Với $r = 1, 2, \ldots, k$, mỗi mệnh đề C_r có chính xác ba trực kiện riêng biệt l_1^r, l_2^r, và l_3^r. Ta sẽ kiến tạo một đồ thị G sao cho ϕ là thỏa mãn được nếu và chỉ nếu G có một bọn có kích cỡ k.

Đồ thị $G = (V, E)$ được kiến tạo như sau. Với mỗi mệnh đề $C_r = (l_1^r \vee l_2^r \vee l_3^r)$ trong ϕ, ta đặt một bộ ba các đỉnh v_1^r, v_2^r, và v_3^r trong V. Ta đặt một cạnh giữa hai đỉnh v_i^r và v_j^s nếu cả hai điều kiện dưới đây tồn tại:

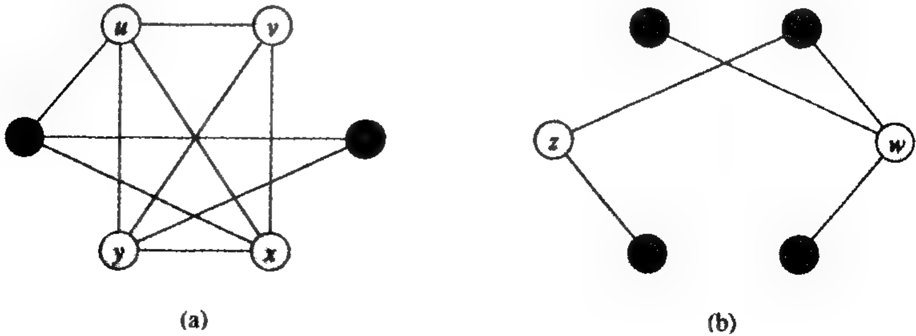
- v_i^r và v_j^s nằm trong các bộ ba khác nhau, nghĩa là, $r \neq s$, và
- Các trực kiện tương ứng của chúng là ***nhất quán***, nghĩa là, l_i^r không phải là phủ định của l_j^s.

Đồ thị này có thể dễ dàng được tính toán từ ϕ trong thời gian đa thức. Để lấy ví dụ về phần kiến tạo này, nếu ta có

$$\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3),$$

thì G là đồ thị nêu trong Hình 36.12.

Ta phải chứng tỏ phép biến đổi này của ϕ thành G là một phép rút gọn. Trước tiên, giả sử rằng ϕ có một phép gán thỏa. Như vậy, mỗi mệnh đề C_r chứa ít nhất một trực kiện l_i^r được gán 1, và mỗi trực kiện như vậy tương ứng với một đỉnh v_i^r. Việc chọn một trực kiện "đúng" như vậy từ mỗi mệnh đề sẽ cho ra một tập hợp V' gồm k đỉnh. Ta biện luận rằng V' là một bọn. Với hai đỉnh bất kỳ $v_i^r, v_j^s \in V'$, ở đó $r \neq s$, cả hai trực kiện tương ứng l_i^r và l_j^s được ánh xạ theo 1 bởi phép gán thỏa đã cho, và như vậy các trực kiện không thể là các phần bù. Do đó, theo phần kiến tạo của G, cạnh (v_i^r, v_j^s) thuộc về E.



Hình 36.13 Rút gọn CLIQUE thành VERTEX-COVER-**(a)** Một đồ thị không hướng $G = (V, E)$ có bọn $V' = \{u, v, x, y\}$. **(b)** Đồ thị $\overline{G}$ được tạo bằng thuật toán rút gọn có vỏ phủ đỉnh $V - V' = \{w, z\}$.

Ngược lại, giả sử G có một bọn V' có kích cỡ k. Không có cạnh nào trong G nối các đỉnh trong cùng một bộ ba, và do đó V' chứa chính xác một đỉnh cho mỗi bộ ba. Ta có thể gán 1 cho mỗi trực kiện l_i^r sao cho $v_i^r \in V'$ mà không sợ phép gán 1 cho một trực kiện lẫn cho phần bù của nó, bởi G không chứa các cạnh giữa các trực kiện không nhất quán. Mỗi mệnh đề được thỏa, và do đó ϕ được thỏa. (Mọi biến không tương ứng với đỉnh trong bọn có thể được ấn định một cách tùy ý.)

Trong ví dụ của Hình 36.12, một phép gán thỏa của ϕ là $<x_1 = 0, x_2 = 0, x_3 = 1>$. Một bọn tương ứng có kích cỡ $k = 3$ bao gồm các đỉnh tương ứng với $\neg x_2$ từ mệnh đề đầu tiên, x_3 từ mệnh đề thứ hai, và x_3 từ mệnh đề thứ ba.

36.5.2 Bài toán vỏ phủ đỉnh

Một ***vỏ phủ đỉnh*** [vertex cover] của một đồ thị không hướng $G = (V, E)$ là một tập hợp con $V' \subseteq V$ sao cho nếu $(u, v) \in E$, thì $u \in V'$ hoặc $v \in V'$ (hoặc cả hai). Nghĩa là, mỗi đỉnh "đề cập" các cạnh liên thuộc của

nó, và một vỏ phủ đỉnh của G là một tập hợp các đỉnh đề cập tất cả các cạnh trong E. Kích cỡ của một vỏ phủ đỉnh là số lượng các đỉnh trong nó. Ví dụ, đồ thị trong Hình 36.13(b) có một vỏ phủ đỉnh $\{w, z\}$ có kích cỡ 2.

Bài toán vỏ phủ đỉnh đó là tìm một vỏ phủ đỉnh có kích cỡ cực tiểu trong một đồ thị đã cho. Để phát biểu lại bài toán tối ưu hóa này dưới dạng một bài toán quyết định, ta muốn xác định xem một đồ thị có một vỏ phủ đỉnh có một kích cỡ đã cho k hay không. Với tư cách là một ngôn ngữ, ta định nghĩa

VERTEX-COVER = $\{<G, k>$: đồ thị G có vỏ phủ đỉnh có kích cỡ $k\}$.

Định lý dưới đây chứng tỏ bài toán này là đầy đủ NP.

Định lý 36.12

Bài toán vỏ phủ đỉnh là đầy đủ NP.

Chứng minh Trước tiên ta chứng tỏ VERTEX-COVER $\in$ NP. Giả sử ta có một đồ thị $G = (V, E)$ và một số nguyên k. Chứng chỉ mà ta chọn là vỏ phủ đỉnh $V' \subseteq V$ chính nó. Thuật toán xác minh khẳng định rằng $|V'| = k$, rồi nó kiểm tra, cho mỗi cạnh $(u, v) \in E$, dẫu $u \in V'$ hoặc $v \in V'$. Phần xác minh này có thể được thực hiện một cách đơn giản trong thời gian đa thức.

Ta chứng minh bài toán vỏ phủ đỉnh là khó NP bằng cách chứng tỏ CLIQUE $\leq_P$ VERTEX-COVER. Phép rút gọn này dựa trên khái niệm về "phần bù" của một đồ thị. Cho một đồ thị không hướng $G = (V, E)$, ta định nghĩa ***phần bù*** [complement] của G là $\overline{G} = (V, \overline{E})$, ở đó $\overline{E} = \{(u, v) : (u, v) \notin E\}$. Nói cách khác, $\overline{G}$ là đồ thị chứa chính xác các cạnh không nằm trong G. Hình 36.13 nêu một đồ thị và phần bù của nó và minh họa phép rút gọn từ CLIQUE thành VERTEX-COVER.

Thuật toán rút gọn nhận một minh dụ $<G, k>$ của bài toán bọn làm đầu vào. Nó tính toán phần bù $\overline{G}$, dễ dàng làm được trong thời gian đa thức. Kết xuất của thuật toán rút gọn là minh dụ $<\overline{G}, |V| - k>$ của bài toán vỏ phủ đỉnh. Để hoàn thành phần chứng minh, ta chứng tỏ phép biến đổi này quả thực là một phép rút gọn: đồ thị G có một bọn có kích cỡ k nếu và chỉ nếu đồ thị $\overline{G}$ có một vỏ phủ đỉnh có kích cỡ $|V| - k$.

Giả sử G có một bọn $V' \subseteq V$ với $|V'| = k$. Ta biện luận rằng $V - V'$ là một vỏ phủ đỉnh trong $\overline{G}$. Cho (u, v) là bất kỳ cạnh nào trong $\overline{E}$. Thì, $(u, v) \notin E$, hàm ý rằng ít nhất một trong số u hoặc v không thuộc về V', bởi mọi cặp đỉnh trong V' được nối bởi một cạnh của E. Theo tương đương, ít nhất một trong số u hoặc v nằm trong $V - V'$, nghĩa là cạnh (u, v) được phủ bởi $V - V'$. Bởi (u, v) đã được chọn một cách tùy ý từ $\overline{E}$, nên mọi

cạnh của $\overline{E}$ được phủ bởi một đỉnh trong $V - V'$. Do đó, tập hợp $V - V'$, có kích cỡ $|V| - k$, sẽ hình thành một vỏ phủ đỉnh cho $\overline{G}$.

Ngược lại, giả sử $\overline{G}$ có một vỏ phủ đỉnh $V' \subseteq V$, ở đó $|V'| = |V| - k$. Thì, với tất cả $u, v \in V$, nếu $(u, v) \in \overline{E}$, thì $u \in V'$ hoặc $v \in V'$ hoặc cả hai. Sự tương phản của phép tất suy này đó là với tất cả $u, v \in V$, nếu $u \notin V'$ và $v \notin V'$, thì $(u, v) \in E$. Nói cách khác, $V - V'$ là một bọn, và nó có kích cỡ $|V| - |V'| = k$.

Bởi VERTEX-COVER là đầy đủ NP, nên ta không dự kiến tìm một thuật toán thời gian đa thức để tìm một vỏ phủ đỉnh có kích cỡ cực tiểu. Tuy nhiên, Đoạn 37.1 trình bày một " thuật toán xấp xỉ" thời gian đa thức, tạo ra các giải pháp "xấp xỉ" cho bài toán vỏ phủ đỉnh. Kích cỡ của một vỏ phủ đỉnh được tạo bởi thuật toán tối đa gấp hai lần kích cỡ cực tiểu của một vỏ phủ đỉnh.

Như vậy, ta đừng nên từ bỏ hy vọng chỉ vì một bài toán là đầy đủ NP. Có thể có một thuật toán xấp xỉ thời gian đa thức đạt được các giải pháp gần tối ưu, cho dù việc tìm một giải pháp tối ưu là đầy đủ NP. Chương 37 cho ta vài thuật toán xấp xỉ cho các bài toán đầy đủ NP.

36.5.3 Bài toán tổng tập con

Bài toán đầy đủ NP kế tiếp mà ta xét mang tính số học. Trong ***bài toán tổng tập con*** [subset-sum], ta có một tập hợp hữu hạn $S \subset \mathbf{N}$ và một ***đích*** $t \in \mathbf{N}$. Ta hỏi có một tập hợp con $S' \subseteq S$ mà các thành phần nó tổng cộng là t hay không. Ví dụ, nếu $S = \{1, 4, 16, 64, 256, 1040, 1041, 1093, 1284, 1344\}$ và $t = 3754$, thì tập hợp con $S' = \{1, 16, 64, 256, 1040, 1093, 1284\}$ là một nghiệm.

Như thường lệ, ta định nghĩa bài toán dưới dạng một ngôn ngữ:

SUBSET-SUM =

$\{<S, t> :$ ở đó tồn tại một tập hợp con $S' \subseteq S$ sao cho $t = \sum_{s \in S'} s\}$.

Như mọi bài toán số học, ta cần nhớ lại rằng phép mã hóa chuẩn mặc nhận các số nguyên đầu vào được mã hóa theo nhị phân. Với giả thiết này trong đầu, ta có thể chứng tỏ bài toán tổng tập con ắt không có một thuật toán nhanh.

Định lý 36.13

Bài toán tổng tập con là đầy đủ NP.

Chứng minh Để chứng tỏ SUBSET-SUM nằm trong NP, với một minh dụ $<S, t>$ của bài toán, ta cho tập hợp con S' là chứng chỉ. Kiểm tra xem $t = \sum_{s \in S'} s$ có thể được hoàn tất bởi một thuật toán xác minh trong thời gian đa thức hay không.

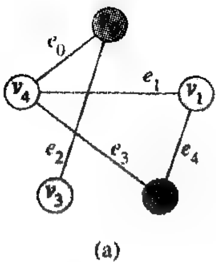
Giờ đây ta chứng tỏ VERTEX-COVER $\leq_p$ SUBSET-SUM. Cho một minh dụ $\langle G, k\rangle$ của bài toán vỏ phủ đỉnh, thuật toán rút gọn kiến tạo một minh dụ $\langle S, t\rangle$ của bài toán tổng tập con sao cho G có một vỏ phủ đỉnh có kích cỡ k nếu và chỉ nếu có một tập hợp con của S mà tổng của nó chính xác là t.

Trọng tâm của phép rút gọn là một phần biểu diễn ma trận liên thuộc của G. Cho $G = (V, E)$ là một đồ thị không hướng và cho $V = \{v_0, v_1, ..., v_{|V|-1}\}$ và $E = \{e_0, e_1, ..., e_{|E|-1}\}$. ***Ma trận liên thuộc*** của G là một ma trận $|V| \times |E|$ $B = (b_{ij})$ sao cho

$$b_{ij} = \begin{cases} 1 \text{ nếu cạnh } e_j \text{ là liên thuộc trên đỉnh } v_i , \\ 0 \text{ bằng không .} \end{cases}$$

Ví dụ, Hình 36.14(**b**) nêu ma trận liên thuộc cho đồ thị không hướng của Hình 36.14(**a**). Ma trận liên thuộc được nêu với các cạnh bên phải có chỉ số dưới, thay vì bên trái như quy ước, để đơn giản hóa các công thức cho các con số trong S.

Cho một đồ thị G và một số nguyên k, thuật toán rút gọn tính toán một tập hợp S các con số và một số nguyên t. Để hiểu rõ cách làm việc của thuật toán rút gọn, ta hãy biểu diễn các con số theo kiểu "cơ số-4 đã sửa đổi." $|E|$ chữ số thứ tự thấp của một số sẽ nằm trong cơ số-4



(a)

	e_4	e_3	e_2	e_1	e_0
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
v_1	1	0	0	1	0
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
v_3	0	0	1	0	0
v_4	0	1	0	1	1

(b)

		cơ số 4 đã sửa đổi							thập phân
			e_4	e_3	e_2	e_1	e_0		
[illegible]									[illegible]
x_1	=	1	1	0	0	1	0	=	1284
[illegible]									[illegible]
x_3	=	1	0	0	1	0	0	=	1040
x_4	=	1	0	1	0	1	1	=	1093
y_0	=	0	0	0	0	0	1	=	1
[illegible]									[illegible]
y_2	=	0	0	0	1	0	0	=	16
y_3	=	0	0	1	0	0	0	=	64
y_4	=	0	1	0	0	0	0	=	256
t	=	3	2	2	2	2	2	=	3754

(c)

Hình 36.14 Phép rút gọn của bài toán vỏ phủ đỉnh thành bài toán tổng tập con. (**a**) Một đồ thị không hướng G. Một vỏ phủ đỉnh $\{v_1, v_3, v_4\}$ có kích cỡ 3 được tô bóng nhạt. (**b**) Ma trận liên thuộc tương ứng. Phần tô bóng các hàng tương ứng với vỏ phủ đỉnh của phần (**a**). Mỗi cạnh e_j có một 1 trong ít nhất một hàng tô bóng nhạt. (**c**) Minh dụ tổng tập hợp con tương ứng. Phần trong hộp là ma trận liên thuộc. Ở đây, vỏ phủ đỉnh $\{v_1, v_3, v_4\}$ có kích cỡ $k = 3$ tương ứng với tập hợp con tô bóng nhạt {1, 16, 64, 256, 1040, 1093, 1284}, cộng dồn tới 3754.

nhưng chữ số thứ tự cao có thể lớn bằng k. Tập hợp các con số được kiến tạo sao cho không có phép mang sang nào có thể lan truyền từ các chữ số thấp hơn đến các chữ số cao hơn.

Tập hợp S bao gồm hai kiểu số, tương ứng với các đỉnh và các cạnh theo thứ tự nêu trên. Với mỗi đỉnh $v_i \in V$, ta tạo một số nguyên dương x_i có phần biểu diễn cơ số-4 đã sửa đổi bao gồm một 1 dẫn đầu theo sau là $|E|$ chữ số. Các chữ số tương ứng với hàng của v_i của ma trận liên thuộc $B = (b_{ij})$ cho G, như minh họa trong Hình 36.14(c). Một cách chính thức, với $i = 0, 1, ..., |V| - 1$,

$$x_i = 4^{|E|} + \sum_{j=0}^{|E|-1} b_{ij}4^j .$$

Với mỗi cạnh $e_j \in E$, ta tạo một số nguyên dương y_j chính là một hàng của ma trận liên thuộc "đồng nhất thức". (Ma trận liên thuộc đồng nhất thức là ma trận $|E| \times |E|$ với chỉ các 1 trong các vị trí đường chéo.) Một cách chính thức, với $j = 0, 1, ..., |E| - 1$,

$$y_j = 4^j .$$

Chữ số đầu tiên của tổng đích t là k, và tất cả $|E|$ chữ số cấp thấp đều là các 2. Một cách chính thức,

$$t = k4^{|E|} + \sum_{j=0}^{|E|-1} 2 \bullet 4^j .$$

Tất cả các con số này có kích cỡ đa thức khi ta biểu diễn chúng theo nhị phân. Phép rút gọn có thể được thực hiện trong thời gian đa thức bằng cách điều tác các bit của ma trận liên thuộc.

Giờ đây, ta phải chứng tỏ đồ thị G có một vỏ phủ đỉnh có kích cỡ k nếu và chỉ nếu có một tập hợp con $S' \subseteq S$ mà tổng của nó là t. Trước tiên, giả sử rằng G có một vỏ phủ đỉnh $V' \subseteq V$ có kích cỡ k. Cho $V' = \{v_{i_1}, v_{i_2}, ..., v_{i_k}\}$, và định nghĩa S' bằng

$$S' = \{x_{i_1}, x_{i_2}, ..., x_{i_k}\} \cup$$
$$\{y_j : e_j \text{ là liên thuộc trên chính xác một đỉnh trong } V'\} .$$

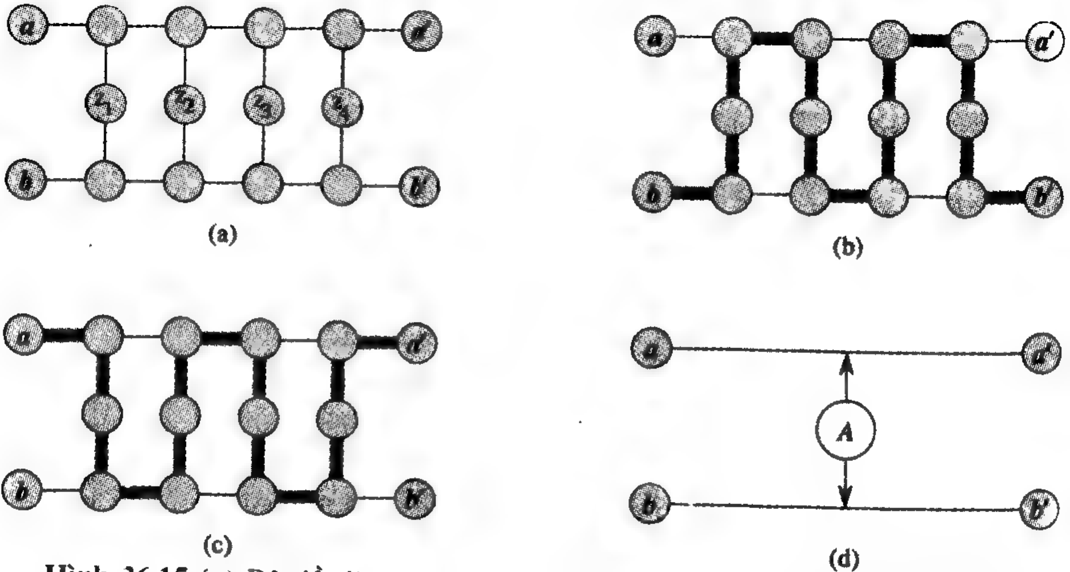
Để xem $\sum_{s \in S'} s = t$, ta nhận thấy rằng việc lấy tổng k 1 dẫn đầu của $x_{i_m} \in S'$ cho ra k chữ số dẫn đầu của phần biểu diễn cơ số-4 đã sửa đổi của t. Để có các chữ số thứ tự thấp của t, mỗi chữ số là một 2, ta lần lượt xét các vị trí chữ số, mỗi vị trí tương ứng với một cạnh e_j. Bởi V' là một vỏ phủ đỉnh, nên e_j liên thuộc trên ít nhất một đỉnh trong V'. Như vậy, với mỗi cạnh e_j, ta có ít nhất một $x_{i_m} \in S'$ với một 1 trong vị trí thứ j. Nếu e_j liên thuộc trên hai đỉnh trong V', thì cả hai đóng góp một 1 vào tổng trong vị trí thứ j. Chữ số thứ j của y_j không đóng góp gì cả, bởi e_j liên thuộc trên hai đỉnh, nên hàm ý rằng $y_j \notin S'$. Như vậy, trong trường hợp này, tổng của S' tạo ra một 2 trong vị trí thứ j của t. Với trường hợp kia—

khi e_j liên thuộc trên chính xác một đỉnh trong V'— ta có $y_j \in S'$, đồng thời đỉnh liên thuộc và y_j đều đóng góp 1 vào tổng của chữ số thứ j của t, và do đó cũng tạo ra một 2. Như vậy, S' là một nghiệm cho minh dụ tổng tập hợp con S.

Giờ đây, giả sử có một tập hợp con $S' \subseteq S$ tổng cộng là t. Cho $S = \{x_{i_1}, x_{i_2}, ..., x_{i_m}\} \cup \{y_{j_1}, y_{j_2}, ..., y_{j_p}\}$. Ta biện luận rằng $m = k$ và rằng $V' = \{v_{i_1}, v_{i_2}, ..., v_{i_m}\}$ là một vỏ phủ đỉnh của G. Để chứng minh biện luận này, ta bắt đầu bằng cách nhận xét rằng với mỗi cạnh $e_j \in E$, có ba 1 trong tập hợp S trong vị trí e_j: một từ mỗi trong số hai đỉnh liên thuộc trên e_j, và một từ y_j. Bởi ta đang làm việc với một phần biểu diễn cơ số-4 đã sửa đổi, nên không có phép mang sang nào từ vị trí e_j đến vị trí e_{j+1}. Như vậy, với mỗi trong số $|E|$ vị trí thứ tự thấp của t, ít nhất là một và tối đa là hai x_i phải đóng góp vào tổng. Bởi ít nhất một x_i đóng góp vào tổng cho mỗi cạnh, ta thấy rằng V' là một vỏ phủ đỉnh. Để xem $m = k$, và do đó V' là một vỏ phủ đỉnh có kích cỡ k, ta nhận thấy cách duy nhất để đạt được k dẫn đầu trong đích t đó là bằng cách gộp chính xác k của x_i trong tổng.

Trong Hình 36.14, vỏ phủ đỉnh $V' = \{v_1, v_3, v_4\}$ tương ứng với tập hợp con $S' = \{x_1, x_3, x_4, y_0, y_2, y_3, y_4\}$. Tất cả y_j đều được gộp trong S', ngoại trừ y_1, là liên thuộc trên hai đỉnh trong V'.

36.5.4 Bài toán chu trình hamilton



Hình 36.15 (a) Bộ đồ dùng A, được dùng trong phép rút gọn từ 3-CNF-SAT thành HAM-CYCLE. **(b)-(c)** Nếu A là một đồ thị con của một đồ thị G chứa một chu trình hamilton và các tuyến nối duy nhất từ A với phần còn lại của G đi qua các đỉnh a, a', b, và b', thì các cạnh tô bóng biểu diễn hai cách duy nhất khả dĩ ở đó chu trình hamilton có thể băng ngang các cạnh của đồ thị con A. **(d)** Một phần biểu diễn nén gọn của bộ đồ dùng A.

Giờ đây, ta trả về bài toán chu trình hamilton được định nghĩa trong Đoạn 36.2.

Định lý 36.14

Bài toán chu trình hamilton là đầy đủ NP.

Chứng minh Trước tiên ta chứng tỏ HAM-CYCLE thuộc về NP. Cho một đồ thị $G = (V, E)$, chứng chỉ của chúng ta là dãy $|V|$ đỉnh tạo thành chu trình hamilton. Thuật toán xác minh sẽ kiểm tra dãy này chứa mỗi đỉnh trong V chính xác một lần và với đỉnh đầu tiên được lặp lại vào cuối, nó hình thành một chu trình trong G. Phần xác minh này có thể được thực hiện trong thời gian đa thức.

Giờ đây ta chứng minh HAM-CYCLE là đầy đủ NP bằng cách chứng tỏ 3-CNF-SAT $\leq_P$ HAM-CYCLE. Cho một công thức Bool ϕ 3-CNF trên các biến $x_1, x_2, \ldots, x_n$ với các mệnh đề $C_1, C_2, \ldots, C_k$, mỗi mệnh đề chứa chính xác 3 trực kiện riêng biệt, ta kiến tạo một đồ thị $G = (V, E)$ trong thời gian đa thức sao cho G có một chu trình hamilton nếu và chỉ nếu ϕ thỏa mãn được. Phần kiến tạo của chúng ta dựa trên các ***bộ đồ dùng*** [widgets], là những mẩu đồ thị áp đặt một số tính chất nhất định.

Bộ đồ dùng đầu tiên của chúng ta là đồ thị con A nêu trong Hình 36.15(a). Giả sử A là một đồ thị con của một đồ thị G và các tuyến nối duy nhất giữa A và phần còn lại của G đi qua các đỉnh a, a', b, và b'. Vả lại, giả sử đồ thị G có một chu trình hamilton. Bởi mọi chu trình hamilton của G phải đi qua các đỉnh z_1, z_2, z_3, và z_4 theo một trong các cách nêu trong các Hình 36.15(b) và (c), nên ta có thể xem đồ thị con A như thể nó đơn giản là một cặp cạnh (a, a') và (b, b') với hạn chế rằng mọi chu trình hamilton của G phải gộp chính xác một trong các cạnh này. Ta sẽ biểu diễn bộ đồ dùng A như đã nêu trong Hình 36.15(d).

Đồ thị con B trong Hình 36.16 là bộ đồ dùng thứ hai. Giả sử rằng B là một đồ thị con của một đồ thị G và các tuyến nối duy nhất từ B với phần còn lại của G đi qua các đỉnh b_1, b_2, b_3, và b_4. Một chu trình hamilton của đồ thị G không thể băng ngang tất cả các cạnh (b_1, b_2), (b_2, b_3), và (b_3, b_4), bởi như vậy tất cả các đỉnh trong bộ đồ dùng khác ngoài b_1, b_2, b_3, và b_4 sẽ bị thiếu. Tuy nhiên, một chu trình hamilton của G có thể băng ngang bất kỳ tập hợp con riêng của các cạnh này. Các Hình 36.16(a)-(e) nêu năm tập hợp con như vậy; hai tập hợp con còn lại có thể có được bằng cách thực hiện một đợt lật từ trên xuống của các phần (b) và (e). Ta biểu diễn bộ đồ dùng này như trong Hình 36.16(*f*), ý tưởng đó là ít nhất một trong các lộ trình được các mũi tên trỏ đến phải được một chu trình hamilton tiếp nhận.

Đồ thị G mà ta sẽ kiến tạo bao gồm hầu hết bản sao của hai bộ đồ

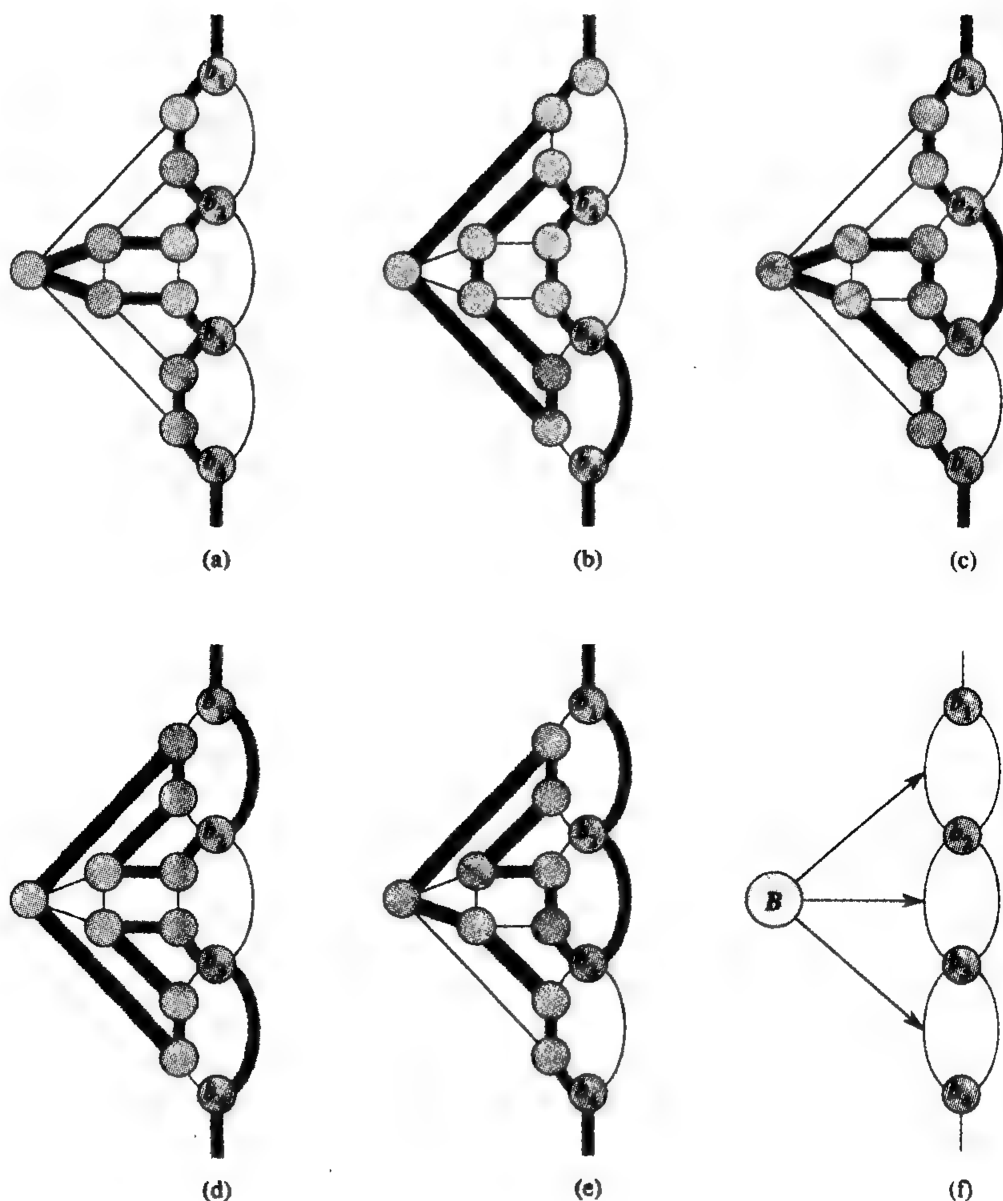
dùng này. Phần kiến tạo được minh họa trong Hình 36.17. Với từng trong số k mệnh đề trong ϕ, ta gộp một bản sao của bộ đồ dùng B, và ta ghép các bộ đồ dùng này với nhau theo chuỗi như sau. Cho b_{ij} là bản sao của đỉnh b_j trong bản sao thứ i của bộ đồ dùng B, ta nối $b_{i,4}$ với $b_{i+1,1}$ với $i = 1, 2,..., k - 1$.

Như vậy, với mỗi biến x_m trong ϕ, ta gộp hai đỉnh x'_m và x''_m. Ta nối hai đỉnh này thông qua hai bản sao của cạnh (x'_m, x''_m), mà ta thể hiện bằng e_m và $\overline{e_m}$ để phân biệt chúng. Ý tưởng đó là nếu chu trình hamilton nhận cạnh e_m, nó tương ứng với việc gán cho biến x_m giá trị 1. Nếu chu trình hamilton nhận cạnh $\overline{e_m}$, biến được gán giá trị 0. Mỗi cặp cạnh này hình thành một vòng lặp hai-cạnh; ta nối các vòng lặp nhỏ này theo chuỗi bằng cách cộng các cạnh (x'_m, x''_{m+1}) với $m = 1, 2,..., n - 1$. Ta nối cạnh trái (mệnh đề) với cạnh phải (biến) của đồ thị thông qua hai cạnh $(b_{1,1}, x'_1)$ và $(b_{k,4}, x''_n)$, là các cạnh trên cùng và dưới cùng trong Hình 36.17.

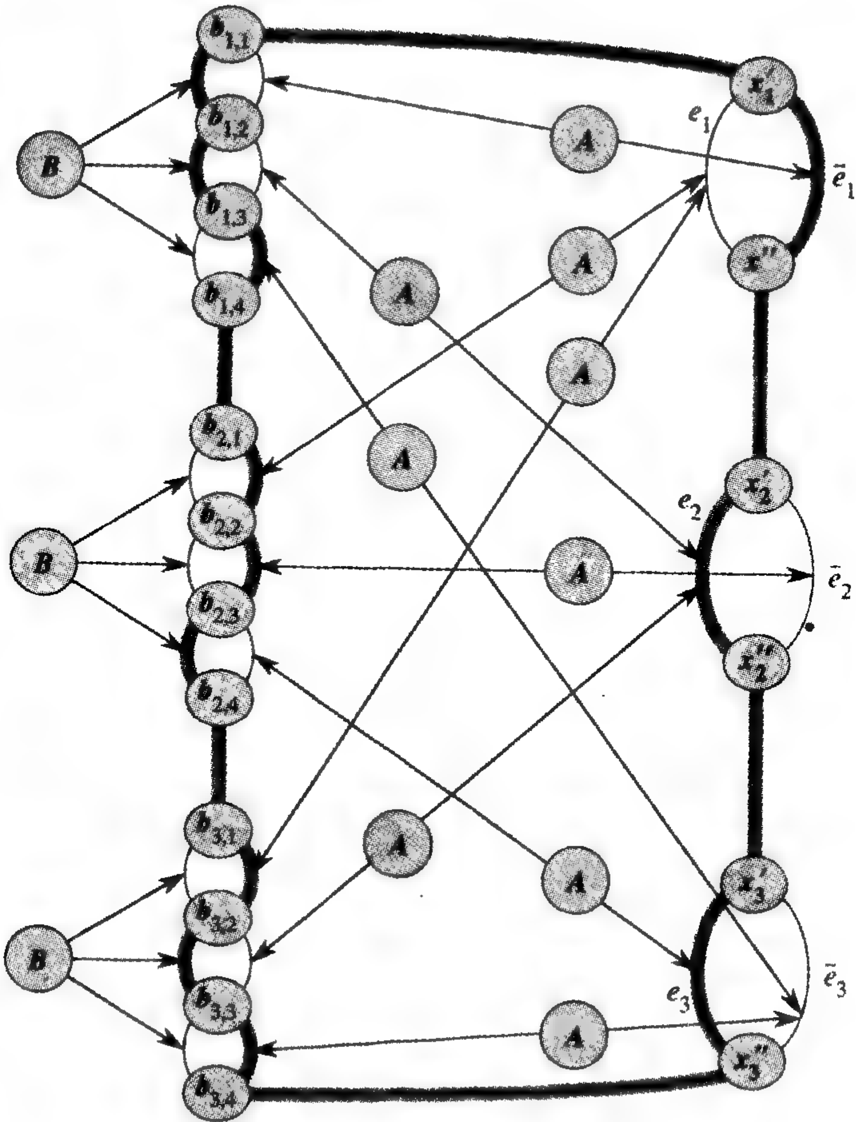
Ta vẫn chưa hoàn tất phần kiến tạo đồ thị G, bởi ta vẫn còn phải liên kết các biến với các mệnh đề. Nếu trực kiện thứ j của mệnh đề C_i là x_m, thì ta dùng một bộ đồ dùng A để nối cạnh $(b_{ij}, b_{i,j+1})$ với cạnh e_m. Nếu trực kiện thứ j của mệnh đề C_i là $\neg x_m$, thì thay vì thế ta đặt một bộ đồ dùng A giữa cạnh $(b_{ij}, b_{i,j+1})$ và cạnh $\overline{e_m}$. Ví dụ, trong Hình 36.17, do mệnh đề C_2 là $(x_1 \vee \neg x_2 \vee x_3)$, nên ta đặt ba bộ đồ dùng A như sau:

- giữa $(b_{2,1}, b_{2,2})$ và e_1,
- giữa $(b_{2,2}, b_{2,3})$ và $\overline{e_2}$, và
- giữa $(b_{2,3}, b_{2,4})$ và e_3.

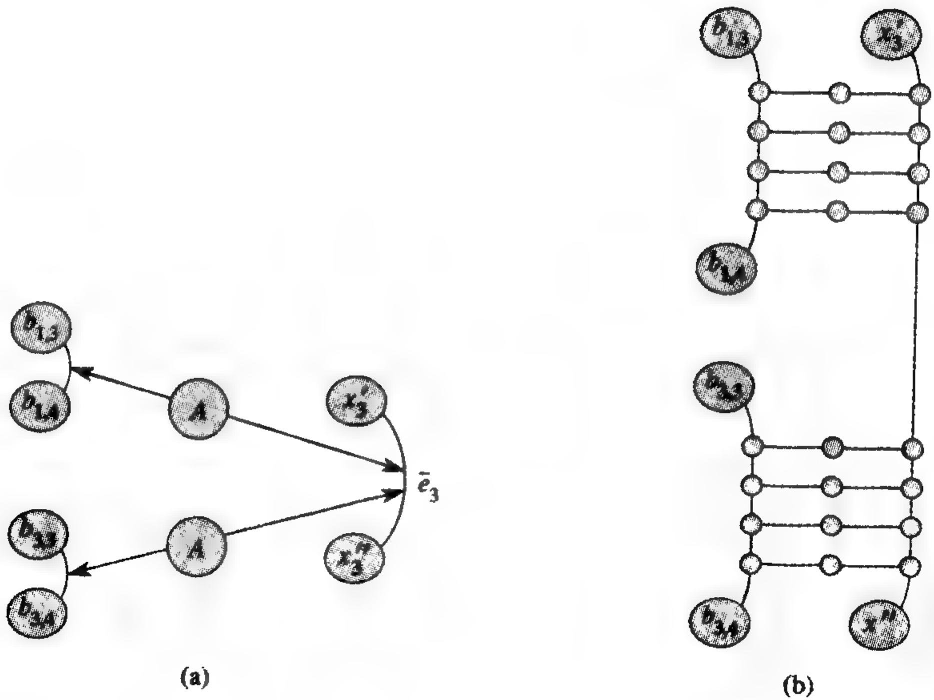
Lưu ý, việc nối hai cạnh thông qua các bộ đồ dùng A thực tế dẫn đến việc thay mỗi cạnh bằng năm cạnh trong đỉnh hoặc đáy của Hình 36.15 (a) và, tất nhiên, cũng bổ sung các tuyến nối đi qua các đỉnh z. Một trực kiện đã cho l_m có thể xuất hiện trong vài mệnh đề (ví dụ, $\neg x_3$ trong Hình 36.17), và như vậy một cạnh e_m hoặc $\overline{e_m}$ có thể bị ảnh hưởng bởi vài bộ đồ dùng A (ví dụ, cạnh $\overline{e_3}$). Trong trường hợp này, ta nối các bộ đồ dùng A theo chuỗi, như đã nêu trong Hình 36.18, thực sự thay cạnh e_m hoặc $\overline{e_m}$ bằng một chuỗi các cạnh.



Hình 36.16 Bộ đồ dùng *B*, được dùng trong phép rút gọn từ 3-CNF-SAT thành HAM-CYCLE. Không có lộ trình nào từ đỉnh b_1 đến đỉnh b_4 chứa tất cả các đỉnh trong bộ đồ dùng có thể dùng cả ba cạnh (b_1, b_2), (b_2, b_3), và (b_3, b_4). Tuy nhiên, có thể dùng mọi tập hợp con riêng của các cạnh này, **(a)**-**(e)** Năm tập hợp con như vậy. **(f)** Một phần biểu diễn của bộ đồ dùng này ở đó ít nhất một trong các lộ trình mà các mũi tên trỏ đến phải được một chu trình hamilton tiếp nhận.



Hình 36.17 Đồ thị G được kiến tạo từ công thức $\phi = (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3)$. Một phép gán thỏa s cho các biến của ϕ là $s(x_1) = 0$, $s(x_2) = 1$, và $s(x_3) = 1$, tương ứng với chu trình hamilton đã nêu. Lưu ý, nếu $s(x_m) = 1$, thì cạnh e_m nằm trong chu trình hamilton, và nếu $s(x_m) = 0$, thì cạnh $\overline{e_m}$, nằm trong chu trình hamilton.



Hình 36.18 Phần kiến tạo thực tế được dùng khi một cạnh e_m hoặc $\overline{e_m}$ bị ảnh hưởng bởi nhiều bộ đồ dùng *A*. **(a)** Một phần của Hình 36.17. **(b)** Đồ thị con thực tế được kiến tạo.

Ta biện luận rằng công thức ϕ là thỏa mãn được nếu và chỉ nếu đồ thị *G* chứa một chu trình hamilton. Trước tiên, ta giả sử *G* có một chu trình hamilton *h* và chứng tỏ ϕ là thỏa mãn được. Chu trình *h* phải có một dạng cụ thể:

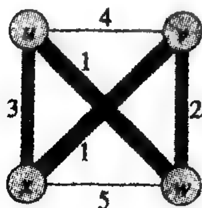
- Trước tiên, nó băng ngang cạnh $(b_{1,1}, x'_1)$ để đi từ đỉnh trái sang đỉnh phải.
- Sau đó, nó theo tất cả x'_m và x''_m đỉnh từ trên xuống, chọn cạnh e_m hoặc cạnh $\overline{e_m}$, nhưng không phải cả hai.
- Kế đó, nó băng ngang cạnh $(b_{k,4}, x''_n)$ để trở về mé trái.
- Cuối cùng, nó băng ngang các bộ đồ dùng *B* từ dưới lên bên trái.

(Nó cũng thực tế băng ngang các cạnh trong các bộ đồ dùng *A*, nhưng ta dùng các đồ thị con này để áp đặt trạng thái tự nhiên hoặc/hoặc của các cạnh mà nó nối.)

Cho chu trình hamilton *h*, ta định nghĩa một phép gán thực cho ϕ như sau. Nếu cạnh e_m thuộc về *h*, thì ta ấn định $x_m = 1$. Bằng không, cạnh $\overline{e_m}$ thuộc về *h*, và ta ấn định $x_m = 0$.

Ta biện luận rằng phép gán này thỏa ϕ. Xét một mệnh đề C_i và bộ

đồ dùng B tương ứng trong G. Mỗi cạnh $(b_{i,j}, b_{i,j+1})$ được nối bằng một bộ đồ dùng A với cạnh e_m hoặc cạnh $\overline{e_m}$, tùy thuộc vào việc x_m hoặc $\neg x_m$ là trực kiện thứ j trong mệnh đề. Cạnh $(b_{i,j}, b_{i,j+1})$ được h băng ngang nếu và chỉ nếu trực kiện tương ứng là 0. Bởi mỗi trong số ba cạnh $(b_{i,1}, b_{i,2})$, $(b_{i,2}, b_{i,3})$, $(b_{i,3}, b_{i,4})$ trong mệnh đề C_i cũng nằm trong một bộ đồ dùng B, nên cả ba không thể được chu trình hamilton h băng ngang. Do đó, một trong ba cạnh phải có một trực kiện tương ứng có giá trị được gán là 1, và mệnh đề C_i được thỏa. Tính chất này áp dụng cho mỗi mệnh đề C_i, $i = 1, 2, ..., k$, và như vậy công thức ϕ được thỏa.



Hình 36.19 Một minh dụ của bài toán người bán hàng du hành. Các cạnh tô bóng biểu diễn một tua có mức hao phí cực tiểu, với mức hao phí 7.

Ngược lại, ta giả sử công thức ϕ được thỏa bằng một phép gán thực. Theo các quy tắc trên đây, ta có thể kiến tạo một chu trình hamilton cho đồ thị G: băng ngang cạnh e_m nếu $x_m = 1$, băng ngang cạnh $\overline{e_m}$ nếu $x_m = 0$, và băng ngang cạnh $(b_{i,j}, b_{i,j+1})$ nếu và chỉ nếu trực kiện thứ j của mệnh đề C_i là 0 dưới phép gán. Các quy tắc này quả thực có thể tuân theo, bởi ta mặc nhận rằng s là một phép gán thỏa cho công thức ϕ.

Cuối cùng, ta lưu ý đồ thị G có thể được kiến tạo trong thời gian đa thức. Nó chứa một bộ đồ dùng B cho mỗi trong số k mệnh đề trong ϕ. Có một bộ đồ dùng A cho mỗi minh dụ của mỗi trực kiện trong ϕ, và do đó có $3k$ bộ đồ dùng A. Bởi các bộ đồ dùng A và B có kích cỡ cố định, nên đồ thị G có $O(k)$ đỉnh và cạnh và được kiến tạo dễ dàng trong thời gian đa thức. Như vậy, ta đã cung cấp một phép rút gọn thời gian đa thức từ 3-CNF-SAT thành HAM-CYCLE.

36.5.5 Bài toán người bán hàng du hành

Trong bài toán người bán hàng du hành, có liên quan mật thiết đến bài toán chu trình hamilton, một người bán hàng phải ghé thăm n thành phố. Lập mô hình bài toán dưới dạng một đồ thị đầy đủ có n đỉnh, ta có thể nói rằng người bán hàng muốn thực hiện một ***tua***, hoặc chu trình hamilton, ghé thăm mỗi thành phố chính xác một lần và kết thúc tại thành phố mà anh ta khởi đầu. Có một mức hao phí số nguyên $c(i, j)$ để du hành từ thành phố i đến thành phố j, và người bán hàng muốn thực hiện tua có tổng mức hao phí là cực tiểu, ở đó tổng mức hao phí là tổng

của các mức hao phí riêng lẻ dọc theo các cạnh của tua. Ví dụ, trong Hình 36.19, tua có mức hao phí cực tiểu là $\langle u, w, v, x, u\rangle$, với mức hao phí 7. Ngôn ngữ hình thức cho bài toán người bán hàng du hành là

TSP = {$\langle G, c, k\rangle$: $G = (V, E)$ là một đồ thị đầy đủ,

c là một hàm từ $V \times V \rightarrow \mathbf{Z}$,

$k \in \mathbf{Z}$, và

G có một tua người bán hàng du hành với mức hao phí tối đa k} .

Định lý dưới đây chứng tỏ một thuật toán nhanh cho bài toán người bán hàng du hành ắt không tồn tại.

Định lý 36.15

Bài toán người bán hàng du hành là đầy đủ NP.

Chứng minh Trước tiên ta chứng tỏ TSP thuộc về NP. Cho một minh dụ của bài toán, ta dùng dãy n đỉnh trong tua làm một chứng chỉ. Thuật toán xác minh sẽ kiểm tra dãy này chứa mỗi đỉnh chính xác một lần, tổng cộng các mức hao phí cạnh, và kiểm tra xem tổng có phải tối đa là k hay không. Tiến trình này chắc chắn có thể được thực hiện trong thời gian đa thức.

Để chứng minh TSP là khó NP, ta chứng tỏ HAM-CYCLE $\leq_P$ TSP. Cho $G = (V, E)$ là một minh dụ của HAM-CYCLE. Ta kiến tạo một minh dụ của TSP như sau. Ta hình thành đồ thị đầy đủ $G' = (V, E')$, ở đó $E' = \{(i, j) : i, j \in V\}$, và ta định nghĩa mức hao phí hàm c bằng

$$c(i, j) = \begin{cases} 0 \text{ nếu } (i, j) \in E, \\ 1 \text{ nếu } (i, j) \notin E. \end{cases}$$

Như vậy, minh dụ của TSP là $(G', c, 0)$, được hình thành dễ dàng trong thời gian đa thức.

Giờ đây, ta chứng tỏ đồ thị G có một chu trình hamilton nếu và chỉ nếu đồ thị G' có một tua có mức hao phí tối đa là 0. Giả sử đồ thị G có một chu trình hamilton h. Mỗi cạnh trong h thuộc về E và như vậy có mức hao phí 0 trong G'. Như vậy, h là một tua trong G' có mức hao phí 0. Ngược lại, giả sử đồ thị G' có một tua h' có mức hao phí tối đa 0. Bởi các mức hao phí của các cạnh trong E' là 0 và 1, mức hao phí của tua h' chính xác là 0. Do đó, h' chỉ chứa các cạnh trong E. Ta kết luận h là một chu trình hamilton trong đồ thị G.

Bài tập

36.5-1

Bài toán đồ thị con đẳng cấu lấy hai đồ thị G_1 và G_2 và hỏi G_1 có phải là một đồ thị con của G_2 hay không. Chứng tỏ bài toán đồ thị con đẳng cấu là đầy đủ NP.

36.5-2

Cho một ma trận số nguyên $m \times n$ A và một m-vectơ số nguyên b, ***bài toán lập trình số nguyên 0-1*** hỏi liệu có một n-vectơ số nguyên x với các thành phần trong tập hợp $\{0,1\}$ sao cho $Ax \leq b$ hay không. Chứng minh kỹ thuật lập trình số nguyên 0-1 là đầy đủ NP. (*Mách nước*: Rút gọn từ 3-CNF-SAT.)

36.5-3

Chứng tỏ bài toán tổng tập con là giải được trong thời gian đa thức nếu giá trị đích t được diễn tả trong đơn phân [unary].

36.5-4

Bài toán tập hợp-phân hoạch nhận một tập hợp S các con số làm đầu vào. Câu hỏi đó là liệu các con số có thể được phân hoạch thành hai tập hợp A và $\overline{A} = S - A$ sao cho $\sum_{x \in A} x = \sum_{x \in \overline{A}} x$ hay không. Chứng tỏ bài toán tập hợp-phân hoạch là đầy đủ NP.

36.5-5

Chứng tỏ bài toán lộ trình hamilton là đầy đủ NP.

36.5-6

Bài toán chu trình đơn giản dài nhất là bài toán xác định một chu trình đơn giản (không có các đỉnh lặp lại) có chiều dài cực đại trong một đồ thị. Chứng tỏ bài toán này là đầy đủ NP.

36.5-7

Giáo sư Marconi cho rằng đồ thị con được dùng làm bộ đồ dùng A trong phần chứng minh của Định lý 36.14 thường phức tạp hơn mức cần thiết: các đỉnh z_3 và z_4 của Hình 36.15(a) và các đỉnh bên trên và bên dưới chúng là không cần thiết. Giáo sư có đúng không? Nghĩa là, phép rút gọn có làm việc với phiên bản nhỏ hơn này của bộ đồ dùng hay không, hoặc giả tính chất "hoặc/hoặc" của bộ đồ dùng có biến mất không?

Các Bài Toán

36-1 Tập hợp độc lập

Một ***tập hợp độc lập*** của một đồ thị $G = (V, E)$ là một tập hợp con $V' \subseteq V$ các đỉnh sao cho mỗi cạnh trong E liên thuộc trên tối đa một đỉnh trong V'. Mục tiêu của ***bài toán tập hợp độc lập*** đó là tìm một tập hợp độc lập có kích cỡ cực đại trong G.

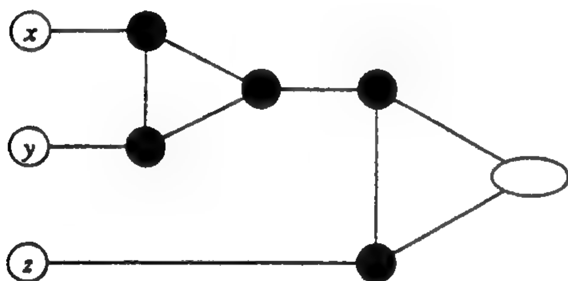
a. Trình bày một bài toán quyết định có liên quan cho bài toán tập hợp độc lập, và chứng minh nó là đầy đủ NP. (*Mách nước*: Rút gọn từ bài toán bọn.)

b. Giả sử bạn được cung cấp một chương trình con để giải bài toán quyết định mà bạn đã định nghĩa trong phần (a). Nêu một thuật toán để tìm một tập hợp độc lập có kích cỡ cực đại. Thời gian thực hiện của thuật toán phải là đa thức trong $|V|$ và $|E|$, ở đó các đợt truy vấn đến hộp đen được đếm dưới dạng một bước đơn.

Mặc dù bài toán quyết định tập hợp độc lập là đầy đủ NP, song một số trường hợp đặc biệt là giải được trong thời gian đa thức.

c. Nêu một thuật toán hiệu quả để giải bài toán tập hợp độc lập khi mỗi đỉnh trong G có độ 2. Phân tích thời gian thực hiện, và chứng minh thuật toán của bạn làm việc đúng đắn.

d. Nêu một thuật toán hiệu quả để giải bài toán tập hợp độc lập khi G là hai nhánh. Phân tích thời gian thực hiện, và chứng minh thuật toán của bạn làm việc đúng đắn. (*Mách nước:* Dùng các kết quả của Đoạn 27.3.)



Hình 36.20 Bộ đồ dùng tương ứng với một mệnh đề $(x \vee y \vee z)$, được dùng trong Bài toán 36-2.

36-2 Tiến trình tô màu đồ thị

Một ***tiến trình tô màu k*** của một đồ thị không hướng $G = (V, E)$ là một hàm $c : V \rightarrow \{1, 2, ..., k\}$ sao cho $c(u) \neq c(v)$ với mọi cạnh $(u, v) \in E$. Nói cách khác, các con số 1, 2,..., k biểu diễn k màu, và các đỉnh kề phải có

các màu khác nhau. Mục tiêu của ***bài toán tiến trình tô màu đồ thị*** đó là xác định số lượng màu cực tiểu cần có để tô một đồ thị đã cho.

a. Nêu một thuật toán hiệu quả để xác định một tiến trình tô màu 2 của một đồ thị nếu như tồn tại.

b. Áp đổi bài toán tiến trình tô màu đồ thị thành một bài toán quyết định. Chứng tỏ bài toán quyết định của bạn có thể giải được trong thời gian đa thức nếu và chỉ nếu bài toán tô màu đồ thị giải được trong thời gian đa thức.

c. Cho ngôn ngữ 3-COLOR là tập hợp các đồ thị có thể được tô màu 3. Chứng tỏ nếu 3-COLOR là đầy đủ NP, thì bài toán quyết định của bạn từ phần (b) là đầy đủ NP.

Để chứng minh 3-COLOR là đầy đủ NP, ta dùng một phép rút gọn từ 3-CNF-SAT. Cho một công thức ϕ gồm m mệnh đề trên n biến $x_1, x_2, \ldots, x_n$, ta kiến tạo một đồ thị $G = (V, E)$ như sau. Tập hợp V bao gồm một đỉnh cho mỗi biến, một đỉnh cho phủ định của mỗi biến, 5 đỉnh cho mỗi mệnh đề, và 3 đỉnh đặc biệt: TRUE, FALSE, và RED. Các cạnh của đồ thị thuộc hai kiểu: các cạnh "trực kiện" độc lập với các mệnh đề và các cạnh "mệnh đề" tùy thuộc vào các mệnh đề. Các cạnh trực kiện hình thành một tam giác trên các đỉnh đặc biệt và cũng hình thành một tam giác trên x_i, $\neg x_i$, và RED với $i = 1, 2, \ldots, n$.

d. Chứng tỏ trong một tiến trình tô màu-3 c của một đồ thị chứa các cạnh trực kiện, có chính xác một cạnh gồm một biến và phủ định của nó được tô màu c(TRUE) và cạnh kia được tô màu c(FALSE). Chứng minh rằng với một phép gán thực bất kỳ với ϕ, ta có một tiến trình tô màu 3 của đồ thị chỉ chứa các cạnh trực kiện.

Bộ đồ dùng nêu trong Hình 36.20 được dùng để áp đặt điều kiện tương ứng với một mệnh đề $(x \vee y \vee z)$. Mỗi mệnh đề yêu cầu một bản sao duy nhất của 5 đỉnh được tô bóng đậm trong hình; chúng nối với các trực kiện của mệnh đề và đỉnh TRUE đặc biệt như đã nêu.

e. Chứng tỏ nếu mỗi trong số x, y, và z được tô màu c(TRUE) hoặc c(FALSE), thì bộ đồ dùng có thể tô màu 3 nếu và chỉ nếu ít nhất một trong số x, y, hoặc z được tô màu c (TRUE).

f. Hoàn tất phần chứng minh 3-COLOR là đầy đủ NP.

Ghi chú Chương

Garey và Johnson [79] cung cấp một phần hướng dẫn tuyệt vời về tính đầy đủ NP, đề cập chi tiết về lý thuyết và cung cấp một catalô về

nhiều bài toán được xem là đầy đủ NP vào năm 1979. (Danh sách các lĩnh vực bài toán đầy đủ NP tại đầu Đoạn 36.5 được rút từ bảng mục lục của họ.) Hopcroft và Ullman [104] và Lewis và Papadimitriou [139] có các cách giải quyết tốt về tính đầy đủ NP trong ngữ cảnh lý thuyết phức. Aho, Hopcroft, va Ullman [4] cũng đề cập đến tính đầy đủ NP và cung cấp vài phép rút gọn, kể cả phép rút gọn cho bài toán vỏ phủ đỉnh từ bài toán chu trình hamilton.

Lớp P đã được Cobham [44] giới thiệu vào năm 1964 và, độc lập, bởi Edmonds [61] vào năm 1965, cũng là người đã giới thiệu lớp NP và đã ước đoán rằng P ≠ NP. Khái niệm về tính đầy đủ NP đã được đề xuất bởi Cook [49] vào năm 1971, là người đã đưa ra các chứng minh tính đầy đủ NP đầu tiên cho khả năng thỏa mãn công thức và khả năng thỏa mãn 3-CNF. Levin [138] đã độc lập khám phá khái niệm, cung cấp một chứng minh tính đầy đủ NP cho một bài toán xếp lát. Karp [116] đã giới thiệu phương pháp luận về các phép rút gọn vào năm 1972 và đã chứng minh nhiều bài toán đầy đủ NP. Tài liệu của Karp có gộp các chứng minh tính đầy đủ NP độc đáo về các bài toán bọn, vỏ phủ đỉnh, và chu trình hamilton. Từ đó, hàng trăm bài toán đã được nhiều nhà nghiên cứu chứng minh là đầy đủ NP.

Phần chứng minh của Định lý 36.14 được thích ứng từ Papadimitriou và Steiglitz [154].

37 Các Thuật Toán Xấp Xỉ

Nhiều bài toán có ý nghĩa thực tiễn là đầy đủ NP nhưng lại quá quan trọng đến nỗi không thể theo đuổi bởi có được một giải pháp tối ưu là điều khó trị. Nếu một bài toán là đầy đủ NP, ta ắt không tìm một thuật toán thời gian đa thức để giải nó chính xác, nhưng điều này không ngụ ý là mất hết hy vọng. Có hai các tiếp cận để khắc phục tính đầy đủ NP. Thứ nhất, nếu các đầu vào thực tế là nhỏ, một thuật toán có thời gian thực hiện hàm mũ có thể hoàn toàn thỏa mãn. Thứ hai, vẫn có thể tìm các giải pháp *gần tối ưu* trong thời gian đa thức (hoặc trong trường hợp xấu nhất hoặc tính trung bình). Trong thực tế, tính gần tối ưu thường là đủ. Một thuật toán trả về các giải pháp gần tối ưu được gọi là một ***thuật toán xấp xỉ***. Chương này trình bày các thuật toán thời gian đa thức xấp xỉ cho vài bài toán đầy đủ NP.

Các cận thực hiện cho các thuật toán xấp xỉ

Giả sử ta đang làm việc trên một bài toán tối ưu hóa ở đó mỗi giải pháp tiềm năng có một mức hao phí dương, và ta muốn tìm giải pháp gần tối ưu. Tùy thuộc vào bài toán, một giải pháp tối ưu có thể được định nghĩa là một giải pháp có mức hao phí khả dĩ cực đại hoặc một giải pháp có mức hao phí khả dĩ cực tiểu; bài toán có thể là một bài toán cực đại hóa hoặc cực tiểu hóa.

Ta nói rằng một thuật toán xấp xỉ cho bài toán có một ***cận tỷ số*** $\rho(n)$ nếu với bất kỳ đầu vào có kích cỡ n, mức hao phí C của giải pháp mà thuật toán xấp xỉ tạo sẽ nằm trong một thừa số $\rho(n)$ của mức hao phí C^* của một giải pháp tối ưu:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \le \rho(n)\,. \qquad (37.1)$$

Định nghĩa này áp dụng cho cả bài toán cực đại hóa lẫn cực tiểu hóa. Với một bài toán cực đại hóa, $0 < C \le C^*$, và tỷ số C^*/C cho ra thừa số qua đó mức hao phí của một giải pháp tối ưu lớn hơn mức hao phí của giải pháp xấp xỉ. Cũng vậy, với một bài toán cực tiểu hóa, $0 < C^* \le C$, và tỷ số C/C^* cho ra thừa số qua đó mức hao phí của giải pháp xấp xỉ lớn hơn mức hao phí của một giải pháp tối ưu. Bởi tất cả các giải pháp được

mặc nhận có mức hao phí dương, các tỷ số này luôn được định nghĩa kỹ. Cận tỷ số của một thuật toán xấp xỉ không bao giờ nhỏ hơn 1, bởi $C/C^* < 1$ hàm ý $C^*/C > 1$. Một thuật toán tối ưu có cận tỷ số 1, và một thuật toán xấp xỉ có một cận tỷ số lớn có thể trả về một giải pháp rất tệ so với tối ưu.

Đôi lúc, để tiện dụng, ta làm việc với một độ đo lỗi tương đối. Với một đầu vào bất kỳ, ***lỗi tương đối*** của thuật toán xấp xỉ được định nghĩa là

$$\frac{|C - C^*|}{C^*},$$

ở đó, như đã nêu, C^* là mức hao phí của một giải pháp tối ưu và C là mức hao phí của giải pháp do thuật toán xấp xỉ tạo ra. Lỗi tương đối luôn không âm. Một thuật toán xấp xỉ có một ***cận lỗi tương đối*** $\epsilon(n)$ nếu

$$\frac{|C - C^*|}{C^*} \le \epsilon(n). \tag{37.2}$$

Nó là do các định nghĩa rằng cận lỗi tương đối có thể được định cận dưới dạng một hàm có cận tỷ số:

$$\epsilon(n) \le \rho(n) - 1. \tag{37.3}$$

(Với một bài toán cực tiểu hóa, đây là một đẳng thức, trong khi đó với bài toán cực đại hóa, ta có $\epsilon(n) = (\rho(n) - 1)/\rho(n)$, thỏa bất đẳng thức (37.3) bởi $\rho(n) \ge 1$.)

Với nhiều bài toán, các thuật toán xấp xỉ đã được phát triển có một cận tỷ số cố định, độc lập với n. Với các bài toán như vậy, ta đơn giản dùng hệ ký hiệu ρ hoặc ϵ, nêu rõ không có sự phụ thuộc trên n.

Với các bài toán khác, các nhà khoa học máy tính đã không thể nghĩ ra các thuật toán xấp xỉ thời gian đa thức có một cận tỷ số cố định. Với các bài toán như vậy, cách tốt nhất đó là để cận tỷ số tăng trưởng dưới dạng một hàm có kích cỡ đầu vào n. Một ví dụ về bài toán như vậy đó là bài toán phủ tập hợp được trình bày trong Đoạn 37.3.

Có vài bài toán đầy đủ NP cho phép các thuật toán xấp xỉ có thể đạt được các cận tỷ số nhỏ hơn dần (hoặc, theo tương đương, các cận lỗi tương đối nhỏ hơn dần) bằng cách sử dụng ngày càng nhiều thời gian tính toán. Nghĩa là, có sự trả giá giữa thời gian tính toán và chất lượng của phép xấp xỉ. Một ví dụ đó là bài toán tổng tập con được nghiên cứu trong Đoạn 37.4. Tình huống này đủ quan trọng đáng dành riêng một tên cho nó.

Một ***lược đồ xấp xỉ*** cho một bài toán tối ưu hóa là một thuật toán xấp

xỉ chấp nhận không những một minh dụ của bài toán, mà còn một giá trị $\epsilon > 0$ làm đầu vào sao cho với bất kỳ ϵ, cố định, lược đồ là một thuật toán xấp xỉ có cận lỗi tương đối ϵ. Ta nói rằng một lược đồ xấp xỉ là một ***lược đồ xấp xỉ thời gian đa thức*** nếu với bất kỳ $\epsilon > 0$ cố định, lược đồ chạy trong thời gian đa thức trong kích cỡ n của minh dụ đầu vào của nó.

Thời gian thực hiện của một lược đồ xấp xỉ thời gian đa thức không được gia tăng quá nhanh khi ϵ giảm. Về lý tưởng, nếu ϵ giảm theo một thừa số bất biến, thời gian thực hiện để đạt được mức xấp xỉ mong muốn không được gia tăng nhiều hơn một thừa số bất biến. Nói cách khác, ta muốn thời gian thực hiện là đa thức trong $1/\epsilon$ cũng như trong n.

Ta nói rằng một lược đồ xấp xỉ là một ***lược đồ xấp xỉ thời gian đa thức đầy đủ*** nếu thời gian thực hiện của nó là đa thức trong cả $1/\epsilon$ lẫn trong kích cỡ n của minh dụ đầu vào, ở đó ϵ là cận lỗi tương đối cho lược đồ. Ví dụ, lược đồ có thể có một thời gian thực hiện $(1/\epsilon)^2 n^3$. Với một lược đồ như vậy, mọi mức giảm thừa số bất biến trong ϵ đều có thể đạt được với một mức tăng thừa số bất biến tương ứng trong thời gian thực hiện.

Khái quát chương

Ba đoạn đầu tiên của chương này trình bày vài ví dụ về các thuật toán xấp xỉ thời gian đa thức cho các bài toán đầy đủ NP, và đoạn cuối trình bày một lược đồ xấp xỉ thời gian đa thức đầy đủ. Đoạn 37.1 bắt đầu bằng một nghiên cứu về bài toán vỏ phủ đỉnh, một bài toán cực tiểu hóa đầy đủ NP có một thuật toán xấp xỉ với một cận tỷ số là 2. Đoạn 37.2 trình bày một thuật toán xấp xỉ với cận tỷ số 2 cho trường hợp của bài toán người bán hàng du hành ở đó hàm hao phí thỏa bất đẳng thức tam giác. Nó cũng chứng tỏ rằng nếu không có bất đẳng thức tam giác, một thuật toán xấp xỉ ϵ không thể tồn tại trừ phi P = NP. Đoạn 37.3 nêu cách dùng một phương pháp tham làm một thuật toán xấp xỉ hiệu quả cho bài toán phủ tập hợp, có được một lớp phủ mà mức hao phí của nó trong trường hợp xấu nhất cũng là một thừa số lôga lớn hơn mức hao phí tối ưu. Cuối cùng, Đoạn 37.4 trình bày lược đồ xấp xỉ thời gian đa thức đầy đủ cho bài toán tổng tập con.

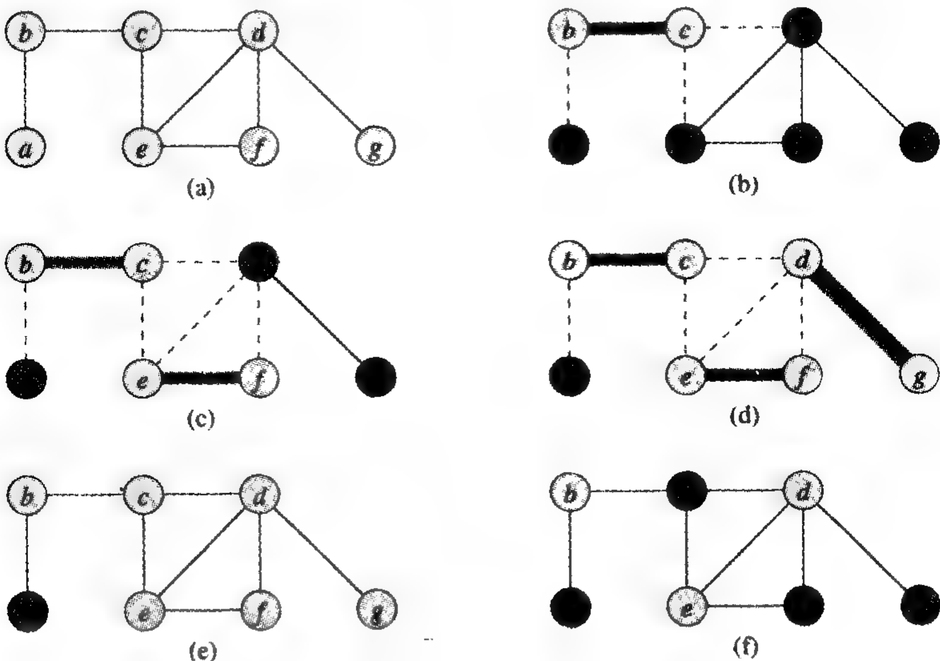
37.1 Bài toán vỏ phủ đỉnh

Bài toán vỏ phủ đỉnh đã được định nghĩa và chứng minh đầy đủ NP trong Đoạn 36.5.2. Một ***vỏ phủ đỉnh*** [vertex cover] của một đồ thị không hướng $G = (V, E)$ là một tập hợp con $V' \subseteq V$ sao cho nếu (u, v) là một

cạnh của G, thì $u \in V'$ hoặc $v \in V'$ (hoặc cả hai). Kích cỡ của một vỏ phủ đỉnh là số lượng các đỉnh trong nó.

Mục tiêu của bài toán vỏ phủ đỉnh đó là tìm ra một vỏ phủ đỉnh có kích cỡ cực tiểu trong một đồ thị không hướng đã cho. Ta gọi một vỏ phủ đỉnh như vậy là một ***vỏ phủ đỉnh tối ưu***. Bài toán này là khó NP, bởi bài toán quyết định có liên quan là đầy đủ NP, theo Định lý 36.12.

Tuy nhiên, cho dù có thể khó lòng tìm ra một vỏ phủ đỉnh tối ưu trong một đồ thị G, quả không quá khó để tìm một vỏ phủ đỉnh gần tối ưu. Thuật toán xấp xỉ dưới đây chấp nhận một đồ thị không hướng G làm đầu vào và trả về một vỏ phủ đỉnh có kích cỡ được bảo đảm không lớn hơn gấp hai lần kích cỡ của một vỏ phủ đỉnh tối ưu.



Hình 37.1 Phép toán của APPROX-VERTEX-COVER. **(a)** Đồ thị đầu vào G, có 7 đỉnh và 8 cạnh. **(b)** Cạnh (b, c), được nêu ở dạng đậm, là cạnh đầu tiên được chọn bởi APPROX-VERTEX-COVER. Các đỉnh b và c, tô bóng nhạt, được bổ sung vào tập hợp A chứa vỏ phủ đỉnh đang được tạo. Các cạnh (a, b), (c, e), và (c, d), gạch cách, được gỡ bỏ bởi giờ đây chúng được phủ bởi một đỉnh trong A. **(c)** Cạnh (e, f) được bổ sung vào A. **(d)** Cạnh (d, g) được bổ sung vào A. **(e)** Tập hợp A, là vỏ phủ đỉnh được APPROX-VERTEX-COVER tạo, chứa sáu đỉnh b, c, d, e, f, g. **(f)** Vỏ phủ đỉnh tối ưu cho bài toán này chỉ chứa ba đỉnh: b, d, và e.

APPROX-VERTEX-COVER(G)

1 $C \leftarrow \emptyset$

```
2  E' ← E[G]
3  while E' ≠ ∅
4       do cho (u, v) là một cạnh tùy ý của E'
5           C ← C ∪ {u, v}
6           gỡ bỏ ra khỏi E' mọi cạnh liên thuộc trên u hoặc v
7  return C
```

Hình 37.1 minh họa phép toán của APPROX-VERTEX-COVER. Biến C chứa vỏ phủ đỉnh đang được kiến tạo. Dòng 1 khởi tạo C theo tập hợp trống. Dòng 2 ấn định E' là một bản sao của tập hợp cạnh $E[G]$ của đồ thị. Vòng lặp trên các dòng 3-6 liên tục chọn một cạnh (u, v) từ E', bổ sung các điểm cuối của nó u và v vào C, và xóa tất cả các cạnh trong E' được phủ bởi u hoặc v. Thời gian thực hiện của thuật toán này là $O(E)$, dùng một cấu trúc dữ liệu thích hợp để biểu thị E'.

Định lý 37.1

APPROX-VERTEX-COVER có một cận tỷ số là 2.

Chứng minh Tập hợp C các đỉnh mà APPROX-VERTEX-COVER trả về là một vỏ phủ đỉnh, bởi thuật toán lặp vòng cho đến khi mọi cạnh trong $E[G]$ được phủ bởi một đỉnh trong C.

Để xem APPROX-VERTEX-COVER trả về một vỏ phủ đỉnh có kích cỡ tối đa gấp hai lần so với một vỏ phủ tối ưu, ta cho A thể hiện tập hợp các cạnh đã được chọn trong dòng 4 của APPROX-VERTEX-COVER. Không có hai cạnh trong A chia sẻ một điểm cuối, bởi một khi một cạnh được chọn trong dòng 4, tất cả các cạnh khác liên thuộc trên các điểm cuối của nó đều được xóa ra khỏi E' trong dòng 6. Do đó, mỗi lần của dòng 5 sẽ bổ sung hai đỉnh mới vào C, và $|C| = 2|A|$. Tuy nhiên, để phủ các cạnh trong A, mọi vỏ phủ đỉnh—nhất là, một vỏ phủ tối ưu C^*—phải gộp ít nhất một điểm cuối của mỗi cạnh trong A. Bởi không có hai cạnh trong A dùng chung một điểm cuối, nên không có đỉnh nào trong vỏ phủ là liên thuộc trên nhiều cạnh trong A. Do đó, $|A| \leq |C^*|$, và $|C| \leq 2|C^*|$, đồng thời chứng minh định lý.

Bài tập

37.1-1

Cho một ví dụ về một đồ thị mà APPROX-VERTEX-COVER luôn cho ra một giải pháp tối ưu con.

37.1-2

Giáo sư Nixon đề xuất heuristic sau đây để giải bài toán vỏ phủ đỉnh. Chọn liên tục một đỉnh có độ cao nhất, và gỡ bỏ tất cả các cạnh liên thuộc của nó. Nêu một ví dụ để chứng tỏ heuristic của giáo sư không có một cận tỷ số 2.

37.1-3

Nêu một thuật toán hiệu quả tham tìm một vỏ phủ đỉnh tối ưu cho một cây trong thời gian tuyến tính.

37.1-4

Từ phần chứng minh của Định lý 36.12, ta biết rằng bài toán vỏ phủ đỉnh và bài toán bọn đầy đủ NP là bù theo nghĩa là một vỏ phủ đỉnh tối ưu là một phần bù của một bọn có kích cỡ cực đại trong đồ thị phần bù. Mối quan hệ này có hàm ý rằng có một thuật toán xấp xỉ với cận tỷ số bất biến cho bài toán bọn hay không? Xác minh đáp án của bạn.

37.2 Bài toán người bán hàng du hành

Trong bài toán người bán hàng du hành đã giới thiệu ở Đoạn 36.5.5, ta có một đồ thị không hướng đầy đủ $G = (V, E)$ có một mức hao phí số nguyên không âm $c(u, v)$ kết hợp với mỗi cạnh $(u, v) \in E$, và ta phải tìm một chu trình hamilton (một tua) của G với mức hao phí cực tiểu. Để mở rộng hệ ký hiệu của chúng ta, cho $c(A)$ thể hiện tổng mức hao phí của các cạnh trong tập hợp con $A \subseteq E$:

$$c(A) = \sum_{(u,v) \in A} c(u, v) .$$

Trong nhiều tình huống thực tiễn, đi trực tiếp từ một nơi u đến một nơi w luôn là cách rẻ nhất; việc đi qua một điểm dừng trung gian v bất kỳ không thể ít tốn kém hơn. Đặt nó theo một cách khác, việc cắt giảm một điểm dừng trung gian không bao giờ gia tăng mức hao phí. Ta hình thức hóa khái niệm này bằng cách phát biểu rằng hàm hao phí c thỏa bất đẳng thức tam giác nếu với tất cả các đỉnh $u, v, w \in V$,

$$c(u, w) \leq c(u, v) + c(v, w) .$$

Bất đẳng thức tam giác là một dạng tự nhiên. và trong nhiều ứng dụng nó tự động được thỏa. Ví dụ. nếu các đỉnh của đồ thị là các điểm trong mặt phẳng và mức hao phí du hành giữa hai đỉnh là khoảng cách euclid bình thường giữa chúng, thì bất đẳng thức tam giác được thỏa.

Như Bài tập 37.2-1 đã nêu, việc hạn chế hàm hao phí để thỏa bất đẳng thức tam giác sẽ không làm thay đổi tính đầy đủ NP của bài toán người bán hàng du hành. Như vậy, không chắc ta có thể tìm ra một thuật toán thời gian đa thức để giải bài toán này một cách chính xác. Do đó thay vì thế, ta tìm các thuật toán xấp xỉ tốt.

Trong Đoạn 37.2.1, ta xét một thuật toán xấp xỉ cho bài toán người bán hàng du hành với bất đẳng thức tam giác có một cận tỷ số là 2. Trong Đoạn 37.2.2, ta chứng tỏ mà không cần bất đẳng thức tam giác, một thuật toán xấp xỉ có cận tỷ số bất biến sẽ không tồn tại trừ phi P = NP.

37.2.1 Bài toán người bán hàng du hành với bất đẳng thức tam giác

Thuật toán sau đây tính toán một tua gần tối ưu của một đồ thị không hướng *G*, dùng thuật toán cây tỏa nhánh cực tiểu MST-PRIM từ Đoạn 24.2. Ta sẽ thấy rằng khi hàm hao phí thỏa bất đẳng thức tam giác, tua mà thuật toán này trả về sẽ không xấu hơn gấp hai lần miễn đó là một tối ưu tua.

APPROX-TSP-TOUR(G, c)

lựa một đỉnh $r \in V[G]$ là một đỉnh "gốc"

tăng trưởng một cây tỏa nhánh cực tiểu T cho G từ gốc r

dùng MST-PRIM(G, c, r)

cho L là danh sách các đỉnh được ghé thăm trong một tầng cây tiền cấp của T

return chu trình hamilton H ghé thăm các đỉnh theo thứ tự L

Hãy nhớ lại từ Đoạn 13.1 rằng một tầng cây tiền cấp ghé thăm đệ quy mọi đỉnh trong cây, liệt kê một đỉnh khi gặp nó lần đầu tiên, trước khi bất kỳ trong số các con của nó được ghé thăm.

Hình 37.2 minh họa phép toán của APPROX-TSP-TOUR. Phần (a) của hình nêu tập hợp các đỉnh đã cho, và phần (b) nêu cây tỏa nhánh

cực tiểu T được MST-PRIM tăng trưởng từ đỉnh gốc a. Phần (c) nêu cách các đỉnh được ghé thăm bởi một tầng tiền cấp của T, và phần (d) hiể › thị tua tương ứng, là tua mà APPROX-TSP-TOUR trả về. Phần (e) hiển thị một tua tối ưu, ngắn hơn khoảng 23%.

Thời gian thực hiện của APPROX-TSP-TOUR là $\Theta(E) = \Theta(V^2)$, bởi đầu vào là một đồ thị đầy đủ (xem Bài tập 24.2-2). Giờ đây ta chứng tỏ nếu hàm hao phí dành cho một minh dụ của bài toán người bán hàng du hành thỏa bất đẳng thức tam giác, thì APPROX-TSP-TOUR trả về một tua có mức hao phí không nhiều hơn gấp hai lần mức hao phí của một tua tối ưu.

Định lý 37.2

APPROX-TSP-TOUR là một thuật toán xấp xỉ có một cận tỷ số là 2 cho bài toán người bán hàng du hành với bất đẳng thức tam giác.

Chứng minh Cho H^* thể hiện một tua tối ưu cho một tập hợp các đỉnh đã cho. Một phát biểu tương đương của định lý đó là $c(H) \leq 2c(H^*)$, ở đó H là tua do APPROX-TSP-TOUR trả về. Bởi ta được một cây tỏa nhánh bằng cách xóa một cạnh bất kỳ ra khỏi một tua, nếu T là một cây tỏa nhánh cực tiểu cho tập hợp các đỉnh đã cho, thì

$$c(T) \leq c(H^*) . \qquad (37.4)$$

Một ***tầng đầy đủ*** [full walk] của T liệt kê các đỉnh khi lần đầu tiên chúng được ghé thăm và mỗi khi chúng được trả về sau một lần ghé thăm một cây con. Ta hãy gọi tầng này là W. Tầng đầy đủ của ví dụ của chúng ta cho thứ tự

$a,b,c,b,h,b,a,d,e,f,e,g,e,d,a.$

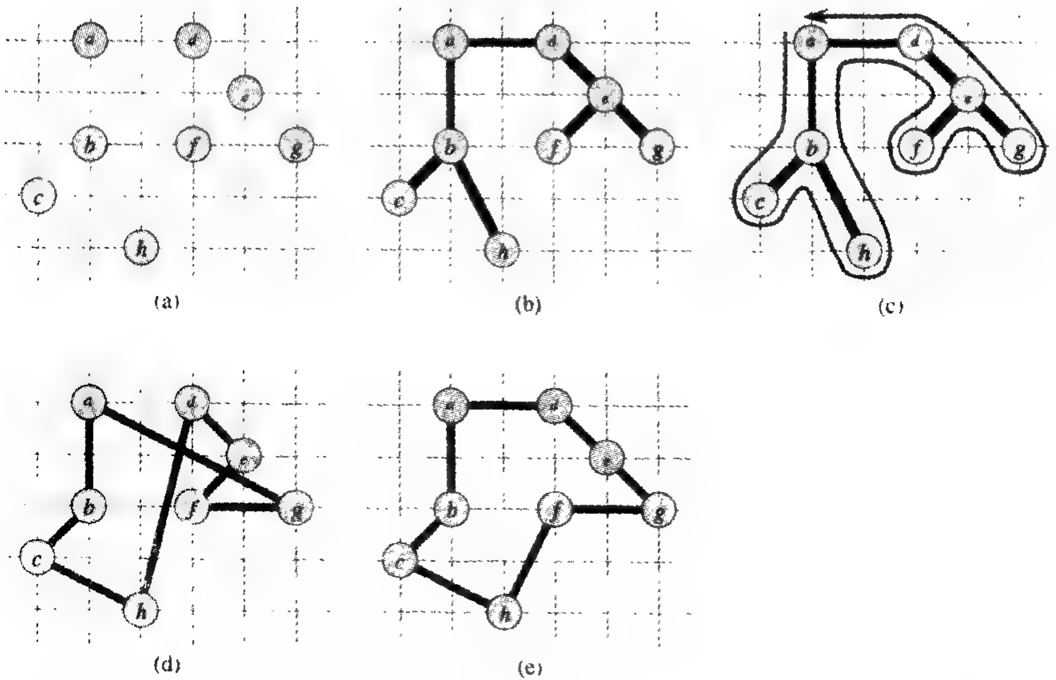
Bởi tầng đầy đủ băng ngang mọi cạnh của T chính xác hai lần, nên ta có

$$c(W) = 2c(T) . \qquad (37.5)$$

Các phương trình (37.4) và (37.5) hàm ý rằng

$$c(W) \leq 2c(H^*) , \qquad (37.6)$$

và do đó mức hao phí của W nằm trong một thừa số của 2 có mức hao phí của một tua tối ưu.



Hình 37.2 Phép toán của APPROX-TSP-TOUR. **(a)** Tập hợp các điểm đã cho, nằm trên các đỉnh của một khung kẻ ô số nguyên. Ví dụ, *f* là một đơn vị về bên phải và hai đơn vị tiến lên từ *h*. Khoảng cách euclid bình thường được dùng làm hàm hao phí giữa hai điểm. **(b)** Một cây tỏa nhánh cực tiểu *T* của các điểm này, như được tính toán bởi MST-PRIM. Đỉnh *a* là đỉnh gốc. Các đỉnh tình cờ được gắn nhãn theo cách chúng được MST-PRIM bổ sung vào cây chính theo thứ tự abc. **(c)** Một tầng của *T*, bắt đầu tại *a*. Một tầng đầy đủ của cây ghé thăm các đỉnh theo thứ tự *a, b, c, b, h, b, a, d, e, f, e, g, e, d, a*. Một tầng tiền cấp của *T* liệt kê một đỉnh ngay khi gặp nó lần đầu tiên, cho ra cách sắp xếp thứ tự *a, b, c, h, d, e, f, g*. **(d)** Một tua các đỉnh có được bằng cách ghé thăm các đỉnh theo thứ tự căn cứ vào tầng tiền cấp. Đây là tua *H* được APPROX-TSP-TOUR trả về. Tổng mức hao phí của nó xấp xỉ là 19.074. **(e)** Một tua tối ưu *H** cho một tập hợp các đỉnh đã cho. Tổng mức hao phí của nó xấp xỉ là 14.715.

Đáng tiếc, thông thường W không phải là một tua, bởi nó ghé thăm vài đỉnh nhiều lần. Tuy nhiên, theo bất đẳng thức tam giác, ta có thể xóa một lần ghé thăm đến một đỉnh bất kỳ từ W và mức hao phí không tăng. (Nếu một đỉnh v được xóa ra khỏi W giữa các lần ghé thăm u và w, cách sắp xếp kết quả chỉ định đi trực tiếp từ u to w.) Bằng cách liên tục áp dụng phép toán này, ta có thể gỡ bỏ ra khỏi W tất cả ngoại trừ lần ghé thăm đầu tiên đến mỗi đỉnh. Trong ví dụ của chúng ta, điều này để lại cách sắp xếp thứ tự

$$a, b, c, h, d, e, f, g .$$

Cách sắp xếp này giống như cách sắp xếp có được bằng một tầng tiền cấp của cây T. Cho H là chu trình tương ứng với tầng tiền cấp này. Nó là một chu trình hamilton, bởi mọi đỉnh được ghé thăm chính xác một lần, và thực tế nó là chu trình đã tính toán bởi APPROX-TSP-TOUR. Bởi H có được bằng cách xóa các đỉnh ra khỏi tầng đầy đủ W, ta có

$$c(H) \leq c(W) . \qquad (37.7)$$

Tổ hợp các bất đẳng thức (37.6) và (37.7) sẽ hoàn tất phần chứng minh.

Bất kể cận tỷ số tế nhị mà Định lý 37.2 cung cấp, APPROX-TSP-TOUR thường không phải là chọn lựa thực tiễn tốt nhất cho bài toán này. Có các thuật toán xấp xỉ khác thường thực hiện ít hơn nhiều trong thực tế (xem các tham chiếu ở cuối chương này).

37.2.2 Bài toán người bán hàng du hành chung

Nếu thải bỏ giả thiết hàm hao phí c thỏa bất đẳng thức tam giác, ta không thể tìm thấy các tua xấp xỉ tốt trong thời gian đa thức trừ phi P = NP.

Định lý 37.3

Nếu P $\neq$ NP và $\rho \geq 1$, không có thuật toán xấp xỉ thời gian đa thức có cận tỷ số ρ cho bài toán người bán hàng du hành chung.

Chứng minh Phần chứng minh là theo sự mâu thuẫn. Vì sự mâu thuẫn, giả sử rằng với một số $\rho \geq 1$, ta có một thuật toán xấp xỉ thời gian đa thức A với cận tỷ số ρ. Không để mất tính tổng quát, ta mặc nhận ρ là một số nguyên, bằng cách làm tròn lên nếu cần. Như vậy, ta sẽ nêu cách sử dụng A để giải các minh dụ của bài toán chu trình hamilton (được định nghĩa trong Đoạn 36.5.5) trong thời gian đa thức. Bởi bài toán chu trình hamilton là đầy đủ NP, theo Định lý 36.14, nên việc giải nó trong thời gian đa thức hàm ý rằng P = NP, theo Định lý 36.4.

Cho $G = (V, E)$ là một minh dụ của bài toán chu trình hamilton. Ta

muốn xác định một cách hiệu quả xem G có chứa một chu trình hamilton hay không bằng cách vận dụng thuật toán xấp xỉ A đã được xây dựng giả thuyết. Ta chuyển G thành một minh dụ của bài toán người bán hàng du hành như sau. Cho $G' = (V, E')$ là đồ thị đầy đủ trên V ; nghĩa là,

$$E' = \{(u,v) : u, v \in V \text{ and } u \neq v\} .$$

Gán một mức hao phí số nguyên cho mỗi cạnh trong E' như sau:

$$c(u, v) = \begin{cases} 1 & \text{nếu } (u, v) \in E , \\ \rho|V| + 1 & \text{bằng không} . \end{cases}$$

Các phần biểu diễn của G' và c có thể được tạo từ một phần biểu diễn của G trong thời gian đa thức trong $|V|$ và $|E|$.

Giờ đây, hãy xét bài toán người bán hàng du hành (G', c). Nếu đồ thị G ban đầu có một chu trình hamilton H, thì hàm hao phí c gán cho mỗi cạnh của H một mức hao phí là 1, và do đó (G', c) chứa một tua có mức hao phí $|V|$. Mặt khác, nếu G không chứa một chu trình hamilton, thì một tua bất kỳ của G' phải dùng một cạnh không nằm trong E. Nhưng bất kỳ tua nào sử dụng một cạnh không nằm trong E đều có một mức hao phí ít nhất là

$$(\rho|V| + 1) + (|V| - 1) > \rho|V| .$$

Bởi các cạnh không nằm trong G thường rất hao phí, nên có một lỗ hổng lớn giữa mức hao phí của một tua là một chu trình hamilton trong G (mức hao phí $|V|$) và mức hao phí của bất kỳ tua nào khác (mức hao phí lớn hơn $\rho|V|$).

Điều gì xảy ra nếu ta áp dụng thuật toán xấp xỉ A cho bài toán người bán hàng du hành (G', c)? Bởi A được bảo đảm để trả về một tua có mức hao phí không nhiều hơn ρ lần so với mức hao phí của một tua tối ưu, nếu G chứa một chu trình hamilton, thì A phải trả về nó. Nếu G không có chu trình hamilton, thì A trả về một tua có mức hao phí lớn hơn $\rho|V|$. Do đó, ta có thể dùng A để giải bài toán chu trình hamilton trong thời gian đa thức.

Bài tập

37.2-1

Nêu trong thời gian đa thức cách thức mà ta có thể biến đổi một minh dụ của bài toán người bán hàng du hành thành một minh dụ khác có hàm hao phí thỏa bất đẳng thức tam giác. Hai minh dụ phải có cùng tập hợp các tua tối ưu. Giải thích tại sao một phép biến đổi thời gian đa thức như vậy không mâu thuẫn với Định lý 37.3, mặc nhận P ≠ NP.

37.2-2

Xét ***heuristic điểm sát nhất*** để xây dựng một tua người bán hàng du hành xấp xỉ. Bắt đầu bằng một chu trình tầm thường bao gồm một đỉnh đơn lẻ được chọn tùy ý. Tại mỗi bước, định danh đỉnh *u* không nằm trên chu trình nhưng có khoảng cách đến một đỉnh bất kỳ trên chu trình là cực tiểu. Giả sử rằng đỉnh trên chu trình là *u* gần nhất chính là đỉnh *v*. Mở rộng chu trình để gộp *u* bằng cách chèn *u* ngay sau *v*. Lặp lại cho đến khi tất cả các đỉnh đều nằm trên chu trình. Chứng minh heuristic này trả về một tua có tổng mức hao phí không nhiều hơn gấp hai lần so với mức hao phí của một tua tối ưu.

37.2-3

Bài toán người bán hàng du hành chỗ nghẽn là bài toán tìm chu trình hamilton sao cho chiều dài của cạnh dài nhất trong chu trình được cực tiểu hóa. Giả sử hàm hao phí thỏa bất đẳng thức tam giác, chứng tỏ ở đó tồn tại một thuật toán xấp xỉ thời gian đa thức có cận tỷ số 3 cho bài toán này. (*Mách nước:* Chứng tỏ theo đệ quy rằng ta có thể ghé thăm tất cả các mắt trong một cây tỏa nhánh cực tiểu chính xác một lần bằng cách lấy một tầng đầy đủ của cây và bỏ qua các mắt, nhưng không bỏ qua trên 2 mắt trung gian liền kề.)

37.2-4

Giả sử rằng các đỉnh cho một minh dụ của bài toán người bán hàng du hành là các điểm trong mặt phẳng và mức hao phí $c(u, v)$ là khoảng cách euclid giữa các điểm *u* và *v*. Chứng tỏ một tua tối ưu không bao giờ đi qua chính nó.

37.3 Bài toán phủ tập hợp

Bài toán phủ tập hợp là một bài toán tối ưu hóa mô hình hóa nhiều bài toán chọn lựa tài nguyên. Nó tổng quát hóa bài toán vỏ phủ đỉnh đầy đủ NP và do đó cũng là khó NP. Tuy nhiên, thuật toán xấp xỉ đã phát triển để điều quản bài toán vỏ phủ đỉnh không áp dụng ở đây, và do đó ta cần thử các cách tiếp cận khác. Ta sẽ xét một heuristic tham đơn giản có một cận tỷ số loga. Nghĩa là, khi kích cỡ của minh dụ trở nên lớn hơn, kích cỡ của giải pháp xấp xỉ có thể tăng trưởng, tương đối với kích cỡ của một giải pháp tối ưu. Tuy nhiên, bởi vì hàm lôga tăng trưởng khá chậm, nên thuật toán xấp xỉ này có thể vẫn cho các kết quả hữu ích.

Một minh dụ $(X, \mathcal{F})$ của ***bài toán phủ tập hợp*** bao gồm một tập hợp

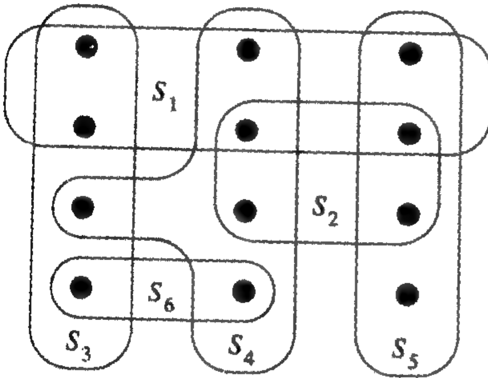
hữu hạn X và một họ $\mathcal{F}$ các tập hợp con của X, sao cho mọi thành phần của X thuộc về ít nhất một tập hợp con trong $\mathcal{F}$:

$$X = \bigcup_{S \in \mathcal{F}} S .$$

Ta nói rằng một tập hợp con $S \in \mathcal{F}$ ***phủ*** các thành phần của nó. Bài toán đó là tìm ra một tập hợp con có kích cỡ cực tiểu $\mathcal{C} \subseteq \mathcal{F}$ có các phần tử phủ tất cả của X:

$$X = \bigcup_{S \in \mathcal{C}} S . \qquad (37.8)$$

Ta nói rằng bất kỳ phương trình thỏa $\mathcal{C}$ (37.8) ***phủ*** X. Hình 37.3 minh họa bài toán.



Hình 37.3 Một minh dụ $(X, \mathcal{F})$ của bài toán phủ tập hợp, ở đó X bao gồm 12 điểm đen và $\mathcal{F} = \{S_1, S_2, S_3, S_4, S_5, S_6\}$. Một vỏ phủ tập hợp có kích cỡ cực tiểu là $\mathcal{C} = \{S_3, S_4, S_5\}$. Thuật toán tham tạo ra một vỏ phủ có kích cỡ 4 bằng cách lựa các tập hợp S_1, S_4, S_5, và S_3 theo thứ tự.

Bài toán phủ tập hợp là một khái niệm trừu tượng của nhiều bài toán tổ hợp thường nảy sinh. Để có ví dụ đơn giản, giả sử rằng X biểu diễn một tập hợp các kỹ năng cần có để giải một bài toán và ta có một tập hợp người đã cho sẵn có để làm việc với bài toán. Ta muốn hình thành một ủy ban, chứa càng ít người càng tốt, sao cho với mọi kỹ thuật cần thiết trong X, ta có một thành viên của ủy ban có kỹ năng đó. Trong phiên bản quyết định của bài toán phủ tập hợp, ta hỏi một lớp phủ có tồn tại với kích cỡ tối đa k hay không, ở đó k là một tham số bổ sung đã chỉ định trong minh dụ bài toán. Phiên bản quyết định của bài toán là đầy đủ NP, như Bài tập 37.3-2 yêu cầu bạn nêu.

Một thuật toán tham xấp xỉ

Phương pháp tham làm việc bằng cách chọn, tại mỗi giai đoạn, tập hợp S phủ hầu hết các thành phần còn lại chưa phủ.

```
GREEDY-SET-COVER(X, 𝓕)
1 U ← X
2 𝒞 ← ∅
3 while U ≠ ∅
4     do lựa một S ∈ 𝓕 tối đa hóa |S ∩ U|
5         U ← U − S
6         𝒞 ← 𝒞 ∪ {S}
7 return 𝒞
```

Trong ví dụ của Hình 37.3, GREEDY-SET-COVER bổ sung vào $\mathcal{C}$ các tập hợp S_1, S_4, S_5, S_3 theo thứ tự.

Thuật toán làm việc như sau. Tại mỗi giai đoạn, tập hợp U chứa tập hợp của các thành phần chưa phủ còn lại. Tập hợp $\mathcal{C}$ chứa vỏ phủ đang được kiến tạo. Dòng 4 là bước ra quyết định tham. Một tập hợp con S được chọn phủ càng nhiều thành phần chưa phủ càng tốt (với các mối ràng buộc được tách một cách tùy ý). Sau khi lựa S, các thành phần của nó được gỡ bỏ ra khỏi U, và S được đặt trong $\mathcal{C}$. Khi thuật toán kết thúc, tập hợp $\mathcal{C}$ chứa một họ con của $\mathcal{F}$ phủ X.

Thuật toán GREEDY-SET-COVER có thể dễ dàng được thực thi để chạy trong thời gian đa thức trong $|X|$ và $|\mathcal{F}|$. Bởi số lượng các lần lặp lại của vòng lặp trên các dòng 3-6 tối đa là $\min(|X|, |\mathcal{F}|)$, và thân vòng lặp có thể được thực thi để chạy trong thời gian $O|X||\mathcal{F}|$), có một thực thi chạy trong thời gian $O(|X||\mathcal{F}| \min(|X|, |\mathcal{F}|))$. Bài tập 37.3-3 yêu cầu một thuật toán thời gian tuyến tính.

Phân tích

Giờ đây ta chứng tỏ thuật toán tham trả về một vỏ phủ tập hợp không quá lớn so với một vỏ phủ tập hợp tối ưu. Để tiện dụng, trong chương này ta thể hiện số điều hòa thứ d $H_d = \sum_{i=1}^{d} 1/i$ (xem Đoạn 3.1) bằng H(d).

Định lý 37.4

GREEDY-SET-COVER có một cận tỷ số

$H(\max \{|S| : S \in \mathcal{F}\})$.

Chứng minh Phần chứng minh tiến hành bằng cách gán một mức hao phí cho mỗi tập hợp đã được thuật toán lựa, phân phối mức hao phí này trên các thành phần được phủ lần đầu tiên, rồi dùng các mức hao phí này để suy ra mối quan hệ mong muốn giữa kích cỡ của một vỏ phủ tập

hợp tối ưu $\mathcal{C}^*$ và kích cỡ của vỏ phủ tập hợp $\mathcal{C}$ mà thuật toán trả về. Cho S_i thể hiện tập hợp con thứ i mà GREEDY-SETCOVER lựa chọn; thuật toán gánh một mức hao phí là 1 khi nó bổ sung S_i vào $\mathcal{C}$. Ta trải đều mức hao phí lựa chọn S_i này giữa các thành phần được phủ lần đầu tiên bởi S_i. Cho c_x thể hiện mức hao phí được phân bổ cho thành phần x, với mỗi $x \in X$. Mỗi thành phần được gán một mức hao phí chỉ một lần, khi nó được phủ lần đầu tiên. Nếu x được phủ lần đầu tiên bởi S_i, thì

$$c_x = \frac{1}{|S_i - (S_1 \cup S_2 \cup \ldots \cup S_{i-1})|}.$$

Thuật toán tìm một nghiệm C của tổng mức hao phí $|\mathcal{C}|$, và mức hao phí này đã được trải rộng trên các thành phần của X. Do đó, bởi vỏ phủ tối ưu $\mathcal{C}^*$ cũng phủ X, nên ta có

$$\begin{aligned} |\mathcal{C}| &= \sum_{x \in X} c_x \\ &\le \sum_{S \in \mathcal{C}^*} \sum_{x \in S} c_x . \end{aligned} \tag{37.9}$$

Phần còn lại của chứng minh dựa trên bất đẳng thức chính dưới đây, mà ta sẽ chứng minh ngắn gọn. Với bất kỳ tập hợp S thuộc về họ $\mathcal{F}$,

$$\sum_{S \in \mathcal{C}^*} c_x \le H(|S|) . \qquad (37.10),$$

Từ các bất đẳng thức (37.9) và (37.10), dẫn đến

$$\begin{aligned} |\mathcal{C}| &\le \sum_{S \in \mathcal{C}^*} H(|S|) \\ &\le |\mathcal{C}^*| \cdot H(\max\{|S| : S \in \mathcal{F}\}) , \end{aligned}$$

chứng minh định lý. Như vậy ta chỉ còn chứng minh bất đẳng thức (37.10). Với bất kỳ tập hợp $S \in \mathcal{F}$ và $i = 1, 2, \ldots, |\mathcal{C}|$, cho

$$u_i = |S - (S_1 \cup S_2 \cup \ldots S_i)|$$

là số lượng các thành phần trong S còn lại chưa phủ sau khi $S_1, S_2, \ldots, S_i$ đã được thuật toán lựa. Ta định nghĩa $u_0 = |S|$ là số lượng các thành phần của S, mà thoạt đầu tất cả đều chưa phủ. Cho k là chỉ số bé nhất sao cho $u_k = 0$, để mọi thành phần trong S được phủ bởi ít nhất một trong số các tập hợp $S_1, S_2, \ldots, S_k$. Như vậy, $u_{i-1} \ge u_i$, và $u_{i-1} - u_i$ thành phần của S được phủ lần đầu tiên bởi S_i, với $i = 1, 2, \ldots, k$. Như vậy,

$$\sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i - (S_1 \cup S_2 \cup \ldots \cup S_{i-1})|}$$

Nhận thấy

$$\begin{aligned} |S_i - (S_1 \cup S_2 \cup \ldots \cup S_{i-1})| &\ge |S - (S_1 \cup S_2 \cup \ldots \cup S_{i-1})| \\ &= u_{i-1} , \end{aligned}$$

bởi sự chọn lựa tham của S_i bảo đảm rằng S không thể phủ các thành phần mới nhiều hơn S_i (bằng không, S đã được chọn thay vì S_i). Bởi vậy, ta được

$$\sum_{x \in S} c_x \leq \sum_{i=1}^{k} (u_{i-1} - u_i) \bullet \frac{1}{u_{i-1}}$$

Với các số nguyên a và b, ở đó $a < b$, ta có

$$\begin{aligned} H(b) - H(a) &= \sum_{i=a+1}^{b} 1/i \\ &\geq (b-a)\frac{1}{b} . \end{aligned}$$

Dùng bất đẳng thức này, ta được tổng lồng gọn [telescoping sum]

$$\begin{aligned} \sum_{x \in S} c_x &\leq \sum_{i=1}^{k} (H(u_{i-1}) - H(u_i)) \\ &= H(u_0) - H(u_k) \\ &= H(u_0) - H(0) \\ &= H(u_0) \\ &= H(|S|) , \end{aligned}$$

bởi $H(0) = 0$. Điều này hoàn tất phần chứng minh của bất đẳng thức (37.10).

Hệ luận 37.5

GREEDY-SET-COVER có một cận tỷ số là $(\ln|X| + 1)$.

Chứng minh Dùng bất đẳng thức (3.12) và Định lý 37.4.

Trong vài ứng dụng, max $\{|S| : S \in \mathcal{F}\}$ là một hằng nhỏ, và do đó giải pháp mà GREEDY-SET-COVER trả về lớn hơn giải pháp tối ưu tối đa một số lần nhỏ bất biến. Một ứng dụng như vậy xảy ra khi heuristic này được dùng để có một vỏ phủ đỉnh xấp xỉ cho một đồ thị mà các đỉnh của nó có độ tối đa là 3. Trong trường hợp này, giải pháp mà GREEDY-SET-COVER tìm thấy không nhiều hơn $H(3) = 11/6$ lần so với một giải pháp tối ưu, một bảo đảm về khả năng thực hiện hơi tốt hơn so với của APPROX-VERTEX-COVER.

Bài tập

37.3-1

Xét từng từ dưới đây dưới dạng một tập hợp các mẫu tự: {arid, dash, drain, heard, lost, nose, shun, slate, snare, thread}. Nêu vỏ phủ tập hợp mà GREEDY-SET-COVER tạo ra khi các mối ràng buộc được tách theo

từ xuất hiện đầu tiên trong từ điển.

37.3-2

Chứng tỏ phiên bản quyết định của bài toán phủ tập hợp là đầy đủ NP bằng phép rút gọn từ bài toán vỏ phủ đỉnh.

37.3-3

Nêu cách thực thi GREEDY-SET-COVER theo cách nó chạy trong thời gian $O(\sum_{S \in \mathcal{F}} |S|)$.

37.3-4

Chứng tỏ dạng yếu hơn dưới đây của Định lý 37.4 là đúng theo tầm thường:

$$|\mathcal{C}| \leq |\mathcal{C}^*| \max\{|S| : S \in \mathcal{F}\}.$$

37.3-5

Tạo một họ các minh dụ vỏ phủ tập hợp chứng minh GREEDY-SET-COVER có thể trả về một số giải pháp khác nhau theo hàm mũ trong kích cỡ của minh dụ. (Các giải pháp khác nhau là kết quả từ các mối ràng buộc đang được tách khác nhau trong sự chọn lựa S trong dòng 4.)

37.4 Bài toán tổng tập con

Một minh dụ của bài toán tổng tập con là một cặp (S, t), ở đó S là một tập hợp $\{x_1, x_2, ..., x_n\}$ các số nguyên dương và t là một số nguyên dương. Bài toán quyết định này hỏi có tồn tại một tập hợp con của S cộng dồn chính xác lên giá trị đích t hay không. Bài toán này là đầy đủ NP (xem Đoạn 36.5.3).

Bài toán tối ưu hóa kết hợp với bài toán quyết định này nảy sinh trong các ứng dụng thực tiễn. Trong bài toán tối ưu hóa, ta muốn tìm một tập hợp con $\{x_1, x_2, ..., x_n\}$ có tổng càng lớn càng tốt nhưng không lớn hơn t. Ví dụ, ta có thể có một xe tải có thể chở không quá t pound, và n hộp khác nhau để chuyên chở, hộp thứ i cân nặng x_i pound. Ta muốn chất lên xe tải càng đầy càng tốt mà không vượt quá giới hạn trọng số đã cho.

Trong đoạn này, ta trình bày một thuật toán thời gian mũ cho bài toán tối ưu hóa này rồi nêu cách sửa đổi thuật toán sao cho nó trở thành một lược đồ xấp xỉ thời gian đa thức đầy đủ. (Hãy nhớ lại một lược đồ xấp xỉ thời gian đa thức đầy đủ có một thời gian thực hiện là đa thức trong $1/\epsilon$ cũng như trong n.)

Một thuật toán thời gian mũ

Nếu L là một danh sách các số nguyên dương và x là một số nguyên dương khác, thì ta cho $L + x$ thể hiện danh sách các số nguyên phái sinh từ L bằng cách tăng mỗi thành phần của L theo x. Ví dụ, nếu $L = \langle 1, 2, 3, 5, 9 \rangle$, thì $L + 2 = \langle 3, 4, 5, 7, 11 \rangle$. Ta cũng dùng hệ ký hiệu này cho các tập hợp, sao cho

$$S + x = \{s + x : s \in S\} .$$

Ta dùng một thủ tục phụ MERGE-LISTS(L, L') trả về danh sách đã sắp xếp được hợp nhất từ hai danh sách đầu vào đã sắp xếp của nó là L và L'. Cũng như thủ tục MERGE mà ta đã dùng trong thuật toán sắp xếp trộn (Đoạn 1.3.1), MERGE-LISTS chạy trong thời gian $O(|L| + |L'|)$. (Ta bỏ qua việc cung cấp mã giả cho MERGE-LISTS.) Thủ tục EXACT-SUBSET-SUM nhận tập hợp $S = \{x_1, x_2, ..., x_n\}$ và một giá trị đích t làm đầu vào.

EXACT-SUBSET-SUM(S, t)

1 $n \leftarrow |S|$

2 $L_0 \leftarrow \langle 0 \rangle$

3 **for** $i \leftarrow 1$ **to** n

4 **do** $L_i \leftarrow$ MERGE-LISTS(L_{i-1}, $L_{i-1} + x_i$)

5 gỡ bỏ ra khỏi L_i mọi thành phần lớn hơn t

6 **return** thành phần lớn nhất trong L_n

Cho P_i thể hiện tập hợp của tất cả các giá trị có thể có được bằng cách lựa một tập hợp con (có thể trống) gồm $\{x_1, x_2, ..., x_i\}$ và tính tổng các phần tử của nó. Ví dụ, nếu $S = \{1, 4, 5\}$, thì

$P_1 = \{0, 1\}$

$P_2 = \{10,1,4,5\}$

$P_3 = \{0, 1, 4, 5, 6, 9, 10\}$.

Cho đồng nhất thức

$$P_i = P_{i-1} \cup (P_{i-1} + x_i) , \qquad (37.11)$$

ta có thể chứng minh theo quy nạp trên i (xem Bài tập 37.4-1) rằng danh sách L_i là một danh sách đã sắp xếp chứa mọi thành phần của P_i có giá trị không lớn hơn t. Bởi chiều dài của L_i có thể nhiều tới mức 2^i, nên EXACT-SUBSET-SUM là một thuật toán thời gian mũ nói chung, mặc dù nó là một thuật toán thời gian đa thức trong các trường hợp đặc biệt ở đó t là đa thức trong $|S|$ hoặc tất cả các con số trong S được định cận bởi một đa thức trong $|S|$.

Một lược đồ xấp xỉ thời gian đa thức đầy đủ

Ta có thể suy ra một lược đồ xấp xỉ thời gian đa thức đầy đủ cho bài toán tổng tập con bằng cách "tỉa" [trimming] mỗi danh sách L_i sau khi nó được tạo. Ta dùng một tham số tỉa δ sao cho $0 < \delta < 1$. ***Tỉa*** một danh sách L theo δ có nghĩa là gỡ bỏ càng nhiều thành phần ra khỏi L càng tốt, sao cho nếu L' là kết quả của tiến trình tỉa L, thì với mọi thành phần y được gỡ bỏ ra khỏi L, ta có một thành phần $z \le y$ vẫn nằm trong L' sao cho

$$\frac{y - z}{y} \le \delta$$

hoặc, theo tương đương,

$$(1 - \delta)y \le z \le y.$$

Ta có thể xem một z như vậy là "biểu thị" cho y trong danh sách mới L'. Mỗi y được biểu thị bởi một z sao cho lỗi tương đối của z, đối với y, tối đa là δ. Ví dụ, nếu $\delta = 0.1$ và

$$L = <10, 11, 12, 15, 20, 21, 22, 23, 24, 29>,$$

thì ta có thể tỉa L để được

$$L' = <10, 12, 15, 20, 23, 29>,$$

ở đó giá trị đã xóa 11 được biểu thị bởi 10, các giá trị đã xóa 21 và 22 được biểu thị bởi 20, và giá trị đã xóa 24 được biểu thị bởi 23. Điều quan trọng cần nhớ đó là mọi thành phần của phiên bản đã tỉa của danh sách cũng là một thành phần của phiên bản ban đầu của danh sách. Việc tỉa một danh sách có thể giảm đáng kể số lượng các thành phần trong danh sách trong khi vẫn giữ một giá trị biểu diễn sát (và hơi nhỏ hơn) trong danh sách cho mỗi thành phần được xóa ra khỏi danh sách.

Thủ tục dưới đây tỉa một danh sách đầu vào $L = <y_1, y_2, \dots, y_m>$ trong thời gian $\Theta(m)$, mặc nhận L được sắp xếp theo thứ tự không giảm. Kết xuất của thủ tục là một danh sách đã tỉa, và đã sắp xếp.

```
TRIM(L, δ)
m ← |L|
L' ← <y>
last ← y_1
for i ← 2 to m
    do if last < (1 − δ)y_i
          then chấp y_i lên cuối L'
               last ← y_i
```

8 **return** L'

Các thành phần của L được quét theo thứ tự tăng, và một số được đặt vào danh sách trả về L' chỉ nếu nó là thành phần đầu tiên của L hoặc giả nó không thể được biểu diễn bởi số mới nhất được đặt vào L'.

Cho thủ tục TRIM, ta có thể kiến tạo lược đồ xấp xỉ như sau. Thủ tục này nhận làm đầu vào một tập hợp $S = \{x_1, x_2, \ldots, x_n\}$ n số nguyên (theo thứ tự tùy ý), một số nguyên đích t, và một "tham số xấp xỉ" ϵ, ở đó $0 < \epsilon < 1$.

APPROX-SUBSET-SUM(S, t, ϵ)

1 $n \leftarrow |S|$

2 $L_0 \leftarrow <0>$

3 **for** $i \leftarrow 1$ **to** m

4   **do** $L_i \leftarrow$ MERGE-LISTS($L_{i-1}, L_{i-1} + x_i$)

5         $L_i \leftarrow$ TRIM($L_i, \epsilon/n$)

6   gỡ bỏ ra khỏi L_i mọi thành phần lớn hơn t

7 cho z là giá trị lớn nhất trong L_n

8 **return** z

Dòng 2 khởi tạo danh sách L_0 là danh sách chỉ chứa thành phần 0. Vòng lặp trong các dòng 3-6 có hiệu ứng tính toán L_i dưới dạng một danh sách đã sắp xếp chứa một phiên bản đã tỉa đúng lúc của tập hợp P_i, với tất cả các thành phần lớn hơn t được gỡ bỏ. Bởi L_i được tạo từ L_{i-1}, nên ta phải bảo đảm tiến trình tỉa lặp lại không đưa ra điểm sai quá nhiều. Dưới đây, ta sẽ thấy rằng APPROX-SUBSET-SUM trả về một phép xấp xỉ đúng nếu như có.

Để lấy ví dụ, giả sử ta có minh dụ

$L = <104, 102, 201, 101>$

với $t = 308$ và $\epsilon = 0.20$. Tham số tỉa δ là $\epsilon/4 = 0.05$. APPROX-SUB-SET-SUM tính toán các giá trị dưới đây trên các dòng đã nêu:

dòng 2 : $L_0 = <0>$,

dòng 4 : $L_1 = <0, 104>$,

dòng 5 : $L_1 = <0, 104>$,

dòng 6 : $L_1 = <0, 104>$,

dòng 4 : $L_2 = <0, 102, 104, 206>$,

dòng 5 : $L_2 = <0, 102, 206>$,

dòng 6 : $L_2 = <0, 102, 206>$,

dòng 4 : L_3 = <0,102, 201, 206, 303, 407> ,

dòng 5 : L_3 = <0, 102, 201, 303, 407> ,

dòng 6 : L_3 = <0, 102, 201, 303> ,

dòng 4 : L_4 = <0, 101,102, 201, 203, 302, 303, 404> ,

dòng 5 : L_4 = <0,101, 201, 302, 404> ,

dòng 6 : L_4 = <0, 101, 201, 302> .

Thuật toán trả về $z = 302$ làm đáp án của nó, hoàn toàn nằm trong $\epsilon = 20\%$ của đáp án tối ưu $307 = 104 + 102 + 101$; thực tế, nó nằm trong 2%.

Định lý 37.6

APPROX-SUBSET-SUM là một lược đồ xấp xỉ thời gian đa thức đầy đủ cho bài toán tổng tập con.

Chứng minh Các phép toán tỉa L_i trong dòng 5 và gỡ bỏ ra khỏi L_i mọi thành phần lớn hơn t sẽ duy trì tính chất rằng mọi thành phần của L_i cũng là một phần tử của P_i. Do đó, giá trị z được trả về trong dòng 8 quả thực là tổng của một tập hợp con của S. Còn lại ta chỉ cần chứng tỏ nó không nhỏ hơn $1 - \epsilon$ lần so với một giải pháp tối ưu. (Lưu ý, bởi bài toán tổng tập con là một bài toán cực đại hóa, nên phương trình (37.2) tương đương với $C^* (1 - \epsilon) \leq C$.) Ta cũng phải chứng tỏ thuật toán chạy trong thời gian đa thức.

Để chứng tỏ lỗi tương đối của đáp án trả về là nhỏ, ta lưu ý khi danh sách L_i được tỉa, ta đưa ra một lỗi tương đối của tối đa ϵ / n giữa các giá trị đại diện còn lại và các giá trị trước khi tỉa. Theo quy nạp trên i, nó có thể được chứng tỏ rằng với mọi thành phần y trong P_i mà tối đa là t, ta có một $z \in L_i$ sao cho

$$(1 - \epsilon/ n)^i y \leq z \leq y . \qquad (37.12)$$

Nếu $y^* \in P_n$ thể hiện một giải pháp tối ưu cho bài toán tổng tập con, thì có một $z \in L_n$ sao cho

$$(1 - \epsilon/ n)^n y^* \leq z \leq y^* ; \qquad (37.13)$$

z lớn nhất như vậy là giá trị mà APPROX-SUBSET-SUM trả về. Bởi nó có thể được chứng minh rằng

$$\frac{d}{dn}\left(1 - \frac{\epsilon}{n}\right)^n > 0 ,$$

hàm $(1 - \epsilon /n)^n$ gia tăng với n, sao cho $n > 1$ hàm ý

$$1 - \epsilon < (1 - \epsilon / n)^n ,$$

và như vậy,

$(1-\epsilon)y^* \leq z$.

Do đó, giá trị z mà APPROX-SUNSET-SUM trả về không nhỏ hơn $1-\epsilon$ lần so với giải pháp tối ưu y^*.

Để chứng tỏ đây là một lược đồ xấp xỉ thời gian đa thức đầy đủ, ta suy ra một cận trên chiều dài của L_i. Sau khi tỉa, các thành phần tiếp theo z và z' của L_i phải có mối quan hệ $z / z' > 1/(1-\epsilon/n)$. Nghĩa là, chúng phải khác nhau theo một thừa số của ít nhất $1/(1-\epsilon/n)$. Do đó, số lượng các thành phần trong mỗi L_i tối đa là

$$\log_{1/(1-\epsilon/n)} t = \frac{\ln t}{-\ln(1-\epsilon/n)} .$$
$$\leq \frac{n \ln t}{\epsilon} ,$$

dùng phương trình (2.10). Cận này là đa thức trong số n giá trị đầu vào đã cho, trong số lượng bit $\lg t$ cần có để biểu diễn t, và trong $1/\epsilon$. Bởi thời gian thực hiện của APPROX-SUBSET-SUM là đa thức trong chiều dài của L_i, nên APPROX-SUBSET-SUM là một lược đồ xấp xỉ thời gian đa thức đầy đủ.

Bài tập

37.4-1

Chứng minh phương trình (37.11).

37.4-2

Chứng minh các phương trình (37.12) và (37.13).

37.4-3

Nêu cách sửa đổi lược đồ xấp xỉ được trình bày trong đoạn này để tìm ra một phép xấp xỉ tốt cho giá trị nhỏ nhất không nhỏ hơn t là một tổng của một tập hợp con của danh sách đầu vào đã cho?

Bài Toán

37-1 Đóng gói giỏ

Giả sử ta có một tập hợp n đối tượng, ở đó kích cỡ s_i của đối tượng thứ i thỏa $0 < s_i < 1$. Ta muốn đóng gói tất cả các đối tượng vào số lượng cực tiểu các giỏ có kích cỡ đơn vị. Mỗi giỏ có thể lưu giữ một tập hợp con bất kỳ các đối tượng có tổng kích cỡ không vượt quá 1.

a. Chứng minh bài toán xác định số lượng cực tiểu các giỏ cần có là

khó NP [NP-hard]. (*Mách nước:* Rút gọn từ bài toán tổng tập con.)

Heuristic ***hứng đầu tiên*** [first-fit] lần lượt lấy từng đối tượng và đặt nó vào giỏ đầu tiên có thể đựng nó. Cho $S = \Sigma_{i=1}^{n} s_i$.

b. Chứng tỏ số lượng tối ưu các giỏ cần có ít nhất là $\lceil S \rceil$

c. Chứng tỏ heuristic hứng đầu tiên để lại tối đa một giỏ ít hơn mức đầy một nửa.

d. Chứng minh số lượng các giỏ được heuristic hứng đầu tiên sử dụng không bao giờ nhiều hơn $\lceil 2S \rceil$.

e. Chứng minh một cận tỷ số 2 cho heuristic hứng đầu tiên.

f. Nêu một kiểu thực thi hiệu quả của heuristic hứng đầu tiên, và phân tích thời gian thực hiện của nó.

37-2 Lấy xấp xỉ kích cỡ của một bọn cực đại

Cho $G = (V, E)$ là một đồ thị không hướng. Với bất kỳ $k \geq 1$, định nghĩa $G^{(k)}$ là đồ thị không hướng $(V^{(k)}, E^{(k)})$, ở đó $V^{(k)}$ là tập hợp của tất cả k-tuple có sắp xếp của các đỉnh từ V và $E^{(k)}$ được định nghĩa sao cho $(v_1, v_2, \ldots, v_k)$ kề với $(w_1, w_2, \ldots, w_k)$ nếu và chỉ nếu với một i, đỉnh v_i kề với w_i trong G.

a. Chứng minh kích cỡ của bọn cực đại [maximum clique] trong $G^{(k)}$ bằng với lũy thừa thứ k của kích cỡ của bọn cực đại trong G.

b. Chứng tỏ nếu có một thuật toán xấp xỉ có một cận tỷ số bất biến để tìm một bọn kích cỡ cực đại, thì ta có một lược đồ xấp xỉ thời gian đa thức đầy đủ cho bài toán.

37-3 Bài toán phủ tập hợp gia trọng

Giả sử ta tổng quát hóa bài toán phủ tập hợp sao cho mỗi tập hợp S_i trong họ $\mathcal{F}$ có một trọng số kết hợp w_i và trọng số của một vỏ phủ $\mathcal{C}$ là $\Sigma_{S_i \in \mathcal{C}} w_i$. Ta muốn xác định một vỏ phủ trọng số cực tiểu. (Đoạn 37.3 điều quản trường hợp ở đó $w_i = 1$ với tất cả i.)

Chứng tỏ heuristic phủ tập hợp tham có thể được tổng quát hóa theo cách tự nhiên để cung cấp một giải pháp xấp xỉ với bất kỳ minh dụ nào của bài toán phủ tập hợp gia trọng [weighted set-covering problem]. Chứng tỏ heuristic có một cận tỷ số là $H(d)$, ở đó d là kích cỡ cực đại của bất kỳ tập hợp S_i.

Ghi chú Chương

Có rất nhiều tài liệu giáo khoa nói về các thuật toán xấp xỉ. Một địa điểm tốt để bắt đầu đó là Garey và Johnson [79]. Papadimitriou và Steiglitz

[154] cũng có một phần trình bày tuyệt vời về các thuật toán xấp xỉ. Lawler, Lenstra, Rinnooy Kan, và Shmoys [133] giải thích chi tiết về bài toán người bán hàng du hành.

Papadimitriou và Steiglitz quy thuật toán APPROX-VERTEX-COVER là của F. Gavril và M. Yannakakis. Thuật toán APPROX-TSP-TOUR xuất hiện trong một tài liệu tuyệt vời của Rosenkrantz, Steams, và Lewis [170]. Định lý 37.3 là của Sahni và Gonzalez [172]. Phần phân tích về heuristic tham cho bài toán phủ tập hợp dựa trên mô hình của phần chứng minh được xuất bản bởi Chvatal [42] có một kết quả chung hơn; kết quả cơ bản này như được trình bày ở đây là của Johnson [113] và Lovasz [141]. Thuật toán APPROX-SUBSET-SUM và phần phân tích của nó phần nào dựa vào các thuật toán xấp xỉ có liên quan cho bài toán ba lô và tổng tập con của Ibarra và Kim [111].

Giáo Trình Thuật Toán
Lý Thuyết và Bài Tập

Sơ Cấp - Trung Cấp - Cao Cấp

Chịu Trách Nhiệm Xuất Bản

CÁT VĂN THÀNH

Biên tập : **NGUYỄN HỮU PHÚ**

Trình bày : **CẨM HỒNG**

Bìa : **ANH THƯ**

Sửa bản in : **NGUYỄN HẢI**

In 1000 cuốn, khổ 16x24cm tại Xưởng In Trung Tâm Hội Chợ Triển Lãm Việt Nam. Giấy phép xuất bản số : 99/XB-QLXB do Cục Xuất Bản cấp ngày 17/01/2001. Trích ngang kế hoạch số : 18-99/XB-QLXB Nhà Xuất Bản Thống Kê cấp ngày 07/08/2001. In xong và nộp lưu chiểu Quý I/2002